KM – The Knowledge Machine 2.0:
Users Manual

Peter Clark
Mathematics and Computing Technology
Boeing Phantom Works
PO Box 3707, Seattle, WA 98124
peter.e.clark@boeing.com

Bruce Porter
Dept of Computer Science
University of Texas at Austin
Austin, TX 78712
porter@cs.utexas.edu

Contents

1 Introduction

1.1 Overview

This manual describes KM, a powerful and mature frame-based knowledge representation language. It is similar in spirit to KRL [1], and has some similarities with KL-ONE representation languages such as LOOM [2] and CLASSIC [3]. Its early origins are from the Theo system [4], and (an earlier version) was the basis for the Botany Knowledge Base project and other KB projects at University of Texas at Austin [5, 6]. It is the reasoning engine used in Shaken, a knowledge acquisition and reasoning system used for several projects including the Rapid Knowledge Formation (RKF) project [7], Vulcan's Project Halo, and Project EPCA, and has also been used for other projects within Boeing and the University of Texas at Austin. Over the years it has evolved to handle a wide variety of representational phenomena, as described in this manual.

In addition to this Users Manual, there is also a Situations Manual [8] describing KM's advanced (but optional) capability for reasoning about different situations, the (Lisp) KM implementation itself, a PowerPoint tutorial, and the example KBs for these manuals, all available at

`http://www.cs.utexas.edu/users/mfkb/km`

Users are also advised to check the KM release notes at this Web address for any updates to KM which go beyond the information described in this manual.

KM is unusual because it continues to work with a frame-based syntax and organization of knowledge. In the 1970's, frame-based representation languages were popular, but many were abandoned in favor of conventional logic syntax because their semantics were imprecise and because they were not expressive enough. In contrast, KM has stayed with the frame-based syntax, and has been extended in multiple ways to accommodate many of the expressiveness requirements that real-world knowledge representation makes, while making sure that its logical semantics remain clearly defined. The result is a logic-based language with an unusual but expressive and, once learned, intuitive syntax for representing and reasoning about the world.

1.2 Features of KM

Before describing the language in detail, we briefly summarize some of the its important features and design goals:

Inference-capable: Perhaps most importantly, KM has been designed to be *inference-capable*, that is to not only represent but also reason with a wide variety of representational forms. Many of the representational features in the language, in particular using frames and access paths, have been chosen with this in mind.

Expressive: As reflected by this manual, KM is highly expressive, and includes support for many fundamental representational phenomena including existential and universal quantification patterns, contexts, defined classes, constraints, intensional representations, situations, and metaclasses.

Formal Semantics: Statements in KM have straightforward and well-defined semantics in (mostly) first-order logic, as described throughout this Manual.

Set-Oriented Evaluation: A technically innovative feature of KM is that it deals with *value sets*, rather than individual values, when finding "the value of a slot" (ie. values for a binary predicate's second argument, given its first). This enables coreferences between statements at different levels of abstraction to be correctly determined (Section 10). For example, if an airplane has two wings, and a 747 airplane has two large wings, KM will heuristically conclude these two wing sets are coreferential, and thus the 747 has two (not four) wings. In contrast, languages such as Prolog which perform "tuple-oriented evaluation", i.e., would 'compute' (ie. search for values for) the wings of a 747 on a value by value basis, often struggle with identifying such coreferences.

Explanations: KM includes a mechanism for producing English-like justifications for its conclusions, using the proof tree supporting those conclusions.

Lazy Unification: To manage the inherent intractability of first-order logic when unifying coreferential instances together, KM performs "lazy unification", meaning it delays computing details of the unified instance until those details are required by some future computation.

Reasoning for Situations: In addition to the features described in this manual, KM supports representation and reasoning about *situations*, where a *situation* can be thought of as a "snapshot" of the world at a particular moment of time, implemented as a separate partition of the knowledge-base which inherits from the main knowledge-base. This capability is an inference-capable implementation of situation calculus, and allows KM to reason about actions and dynamic worlds. It is described described separately in the Situations Manual [8].

Mature: KM has evolved and been used over a long period, and thus it is a relatively mature language, including support for tracing and debugging, and user documentation.

Free: KM is available, including the source code, under the GNU Lesser General Public License (but without warranty). To see a copy of the license, type `(license)` at the KM or Lisp prompt. Further information on the GNU LGPL is also available at http://www.gnu.org/copyleft/lesser.html.

The reader should also note that (like all expressive knowledge representation languages) KM's inference is not logically complete. A discussion of the areas of inferential incompleteness is given in Section 30.

KM is implemented in Common Lisp, and has been tested under Allegro Lisp, (available on both Sun and PC platforms), Harlequin Lisp, and Mac CommonLisp (for the Apple Macintosh).

KM runs under Allegro (ANSI) Common Lisp (alisp), but not under the case-sensitive Allegro Modern Lisp (mlisp). KM does not run under some versions of GNU Lisp due to a GNU implementation bug, in which the function `readtable-case` is not visible in the user package as required by the ANSI Lisp standard.

1.3 Representation and Reasoning in KM

Syntactically, the basic representational unit in KM is a *frame*, which contains *slots* and *values*, for example as shown in Figure 1. Values may be atomic, or may be *expressions* which refer to other frames. Semantically, a frame encodes a set of Horn-clause axioms in order-sorted logic, where frames are sorts (types,classes) or instances, slots are binary relations, and values are the bodies of the Horn-clauses, describing how to compute values for the relation's second argument given the first. A frame can be thought of as a convenient grouping of axioms sharing the same type restriction for their head predicates' first argument (very loosely, grouping all those axioms "about" a concept).

To access information in the knowledge base (KB), the user (or an application system) issues a query to the KB – this may be a request for the value of a slot, or may be a more sophisticated expression. A query interpreter answers the query by evaluating the query expression, to find which value(s) the expression refers to. If those value(s) found are themselves expressions, they in turn are evaluated etc. In this way, the query interpreter performs inference. The language for slot-value expressions and the language for queries are the same, and is described in this document.

Questions are typically posed and answered in terms of instances. For example, the query "what is the weight of a car?" will be answered by first creating an instance of a car, and then computing the filler(s) of its "weight" slot. If the filler is atomic (an instance, number, or string) then it is returned. If it is an expression (for example, informally, "the weight of the engine + the weight of the body"), then it is (recursively) evaluated and the result returned. As these instances are 'anonymous' (ie. here do not refer to any specific car, but instead could be any car), any conclusions will thus apply to *all* instances with the same description. In this example, if the car's weight was computed as 1 ton, then it follow that (according to the KB) all instances of car will have this weight.

The set of instances created during query answering is a (partial) model of the KB plus query; as KM's reasoning is deductive, anything true in this model will be true of *all* models of the KB plus query. This form of inferencing is similar to tableaux inference methods, using deterministic construction rules only [9].

1.4 KM's Syntax

A KM knowledge base consists of a set of axioms in typed, (mainly) first-order logic. However for comprehensibility KM does not use a standard FOL notation (eg. KIF), and instead uses its own syntax described in this manual (and as illustrated in Figure 1). This is intended to make the knowledge base more natural to author, and more comprehensible. In addition, the overall organization of knowledge into a frame hierarchy provides important indexing properties, making inference focused and efficient. Although the syntax here may appear unusual, it is extremely similar to that used in the earlier AI frame language KRL [1], and also has some similarities to other KL-ONE-style languages, suggesting a certain naturalness to this way of expressing knowledge. In addition, it is important to remember that the language is "syntactic sugar" for, rather than a replacement of, first-order logic. A KM knowledge-base could be automatically translated into a more standard notation (eg. KIF).

A disadvantage of using a non-standard syntax is that particular care is needed to ensure that the semantics of expressions are clear and well-understood. In fact, at several points during KM's history we carefully considered changing to a more standard syntax (eg. KIF), to remove possible confusions as to what the KM notation meant. However, the penalty in cumbersomeness was often severe, and so we have instead continued with the notation presented here. Figure 1 shows a KM frame and its equivalent in a more standard notation, to illustrate the difference in comprehensibility for one example.

2 Getting Started

2.1 Loading and Compiling KM

To start the Lisp KM interpreter, start Lisp and then load the `km.lisp` file:

"All buy events have

- a buyer and a seller (both of type agent)
- an object which is bought
- some money equal to the cost of the object
- two 'give' subevents, in which:
    1. The buyer gives the money to the seller
    2. The seller gives the object to the buyer."

(a) English

```
(every Buy has
  (buyer ((a Agent)))
  (object ((a Thing)))
  (seller ((a Agent)))
  (money ((the cost of (the object of Self))))
  (subevent1 ((a Give with
               (agent ((the buyer of Self)))
               (object ((the money of Self)))
               (rcpt ((the seller of Self))))))
  (subevent2 ((a Give with
               (agent ((the seller of Self)))
               (object ((the object of Self)))
               (rcpt ((the buyer of Self)))))))
```

(b) Axiomatization (KM syntax)

```
(forall (?b) (exists (?a) (=> (isa ?b Buy) (and (buyer ?b ?a) (isa ?a Agent)))))
(forall (?b) (exists (?a) (=> (isa ?b Buy) (and (object ?b ?a) (isa ?a Thing)))))
(forall (?b) (exists (?a) (=> (isa ?b Buy) (and (seller ?b ?a) (isa ?a Agent)))))
(forall (?b ?o ?a) (=> (and (isa ?b Buy) (object ?b ?o) (cost ?o ?m)) (money ?b ?m))))
(forall (?b) (exists (?e) (=> (isa ?b Buy) (and (subevent1 ?b ?e) (isa ?e Give)))))
(forall (?b ?e ?a) (=> (and (isa ?b Buy) (subevent1 ?b ?e) (buyer ?b ?a)) (agent ?e ?a))))
(forall (?b ?e ?m) (=> (and (isa ?b Buy) (subevent1 ?b ?e) (money ?b ?m)) (object ?e ?m))))
(forall (?b ?e ?a) (=> (and (isa ?b Buy) (subevent1 ?b ?e) (seller ?b ?a)) (rcpt ?e ?a))))
(forall (?b) (exists (?e) (=> (isa ?b Buy) (and (subevent2 ?b ?e) (isa ?e Give)))))
(forall (?b ?e ?a) (=> (and (isa ?b Buy) (subevent2 ?b ?e) (seller ?b ?a)) (agent ?e ?a))))
(forall (?b ?e ?m) (=> (and (isa ?b Buy) (subevent2 ?b ?e) (object ?b ?m)) (object ?e ?m))))
(forall (?b ?e ?a) (=> (and (isa ?b Buy) (subevent2 ?b ?e) (buyer ?b ?a)) (rcpt ?e ?a))))
```

(c) Axiomatization (KIF syntax)

Figure 1: An example of KM syntax, and it's equivalent in KIF syntax.

```
% lisp                             ; or however you start Lisp
> (load "~/lisp/km")               ; or wherever KM is located
>
```

Note that KM will run considerably faster if compiled. To compile the source, do as a one-time action:

```
% lisp
> (load "km.lisp")                 ; <== you *MUST* do this before compiling!
> (compile-file "km.lisp")
> (load "km")
```

Before compiling you *must* first load `km.lisp`, so that the dispatch macro `#$` (defined in `km.lisp`) is recognized by Lisp during compilation.

Queries are issued to the knowledge-base from a query prompt. The query interpreter performs a read-eval-print loop when processing queries. To get to the query prompt, type

```
> (km)
KM>
```

where "`KM>`" is the query prompt, and from here you can issue queries:

```
KM> (a Car)                     ; create an instance of a car
(_Car1)
```

To exit from the query prompt, type '`q`':

```
KM> q
>
```

To exit from Lisp, type `(quit)`:

```
CL-USER> (quit)
%
```

If for some reason an error occurs and a Lisp debugging prompt appears, you can return to the top level Lisp by typing `:A` (in Harlequin Lisp) or `CTRL-D` (in Allegro Lisp):

```
CL-USER> (/ 1 0)
>>Error: Attempt to divide by zero: (/ 1 0).
:A  0: Abort to Lisp Top Level
-> :A
CL-USER>
```

Finally, note KM is case-sensitive, thus `parent` and `Parent`, say, are considered different when entered at the KM prompt.

2.2 Authoring and Loading KBs

A knowledge-base can either be entered frame by frame at the `KM>` query prompt, or loaded from a file. (Loading a file is equivalent to typing the contents manually at the `KM>` prompt). When authoring in a file, you will need an editor (eg. emacs) open to enter the KB, and a shell running the Lisp KM interpreter to perform inference over the KB. A convenient approach is to start emacs, then split the screen so one buffer is running a Unix shell[1], and the other is editing a text file, which will contain the KB.

The command `(load-kb "`*filename*`")` loads a KB file, and adds the file's contents to the existing KB in memory (thus a KB can be split over multiple files). The command `(reset-kb)` deletes the current KB in memory. The command `(reload-kb "`*filename*`")` does a `(reset-kb)` followed by a `(load-kb "`*filename*`")`. Further documentation on incremental loading of KBs is given later in Section 27.1.

Example KBs including those in this user manual are contained, in a concatenated form, in the file `test-suite.km`, available with the KM source and these manuals. This file gives many examples of KM in use.

`load-kb` has a 'verbose' option allowing you to view the processing of the commands in a file, and can be used as is illustrated below:

```
KM> (load-kb "myfile.km" :verbose t)
```

Note that a KB file can thus contain a test set of queries, as well as KB assertions, or both together. This is useful if you wish to maintain and regularly execute a suite of test queries for testing a KB under construction.

2.3 Comments

Comments can be added to a KB file using the normal Lisp notation for comments:

```
#| Hash-bar delimeters denote multi-line comments,
   as is illustrated here.                          |#

(Car has                          ; one or more semicolons denote
  (superclasses (Vehicle)))       ; a comment on a single line
```

3 Instances and Classes

3.1 Introduction

Just as in the object-oriented paradigm, there are two fundamental types of concepts in KM: *instances* (individuals) and *classes* (types of individuals). A class has an associated collection (its extension, namely individuals in that class), but is distinct from that collection (two concepts, eg. "unicorns" and "dragons", may have the same extension but be different concepts). Properties of members of a class are described by expressions of the form:

```
(every <class> has
    (<slot1> (<expr11> <expr12> ...))
    (<slot2> (<expr21> <expr22> ...))
    ... )
```

---

[1] To start a shell, type `ESC-x shell` in emacs.

where `class` is the name of the class of interest, and the $\texttt{slot}_i$ are binary relations which may hold between instances of `class` and other instances. The $\texttt{expr}_{ij}$ are KM expressions which evaluate to zero or more instances. They can be viewed as functions, parameterized by the instance of `class` being queried, mapping an instance of `class` to a set of values. The $\texttt{slot}_i$ relation is deemed to hold between all instances of `class` and the value(s) that $\texttt{expr}_{ij}$ returns, as shown by this axiom schema for some *class*:

$$\forall x\ isa(x, class) \rightarrow \forall y \in expr_{ij}(x) \quad slot_i(x, y)$$

where $expr_{ij}(x)$ is a function which computes the value of $<\texttt{expr}_{ij}>$ for some instance x of *class*.

Properties of instances are described in a similar way, but omitting the `every` keyword:

```
(<instance> has
    (instance-of (<class1> ... <classn>))
    (<slot1> (<expr11> <expr12> ...))
    (<slot2> (<expr21> <expr22> ...))
    ... )
```

Here, the $\texttt{slot}_i$ relation is deemed to hold between `instance` and the value(s) that $\texttt{expr}_{ij}$ returns:

$$\forall y \in expr_{ij}(I) \ \ slot_i(I, y)$$

where $expr_{ij}(I)$ is a function which computes the value of $<\texttt{expr}_{ij}>$ for I.

Properties of a class itself (rather than its members) use the same syntax as for instances, as these properties treat the class itself as an instance (of the metaclass `Class`). These properties do not apply to the classes' members (operationally speaking, they can be thought of as 'non-inheriting' properties). The most important such properties are `subclasses` and `superclasses`. Other examples include cardinality, average-age, etc. The general form of such expressions is:

```
(<class> has
   (superclasses (<superclass1> ... <superclassN>))
       (<slot1> (<expr11> <expr12> ...))
       (<slot2> (<expr21> <expr22> ...))
       ... )
```

where the `<superclassi>` are the direct superclasses of `<class>`. Thus a class (eg. `Car`) will typically use two frames to describe it, one describing its properties, and one describing its members' properties, eg:

```
;;; "Cars are vehicles."
;;; superclasses(Car,Vehicle) ∧ terms(Car,"car") ∧ terms(Car,"vehicle")
(Car has                          ; properties of the class
  (superclasses (Vehicle))
  (terms ("car" "automobile"))) ; ie. words of phrases used to denote cars

;;; "Every car has four wheels."
;;; ∀c isa(c,Car) → wheels(c,4)
(every Car has                    ; properties of the class's instances
  (wheels (4)))
```

This contrasts with the predecessor of KM (Theo [4]) where all slots were placed on a single frame, and then were flagged as "member" slots (properties of its instances) or "owns" slots (properties of the class itself). Although having two frames per class is syntactically a bit more cumbersome, it is semantically much cleaner, simplifying the inference engine and some notational conventions. In addition, meta-classes can now be used if desired (Section 29.12).

If no superclasses are specified for a class, then the most general class `Thing` is assumed. The inverse `subclasses` links do not need to be explicitly declared in a KB, and are automatically installed for all `superclasses` links encountered. The KM function function `(install-all-subclasses)` will recompute subclass links, removing redundancies and adding links from `Thing` as appropriate.

Whether a concept should be an instance or a class is a modeling decision, depending on which objects are to be considered identical in the representation. For example, we might state a car's color is the instance `Red` (thus modeling it as having the same color as all other objects of color `Red`), or alternatively state the car's color is an instance of the class `Red`. In the former case, KM will consider "two red cars" to have the same color (`Red`), in the latter KM will consider them to have two different colors (eg. `_Red12` and `_Red13`), eg. subtly different shades. KM's notation allows it to tell whether `Red` is being used as an instance or a class; of course, the user must be consistent, and not use the same symbol `Red` to mean an instance in one place and a class in another.

3.2 Example

In a new text file (which will be a new knowledge base), type the following frames below:

```
;;; "Cars are vehicles"
;;; superclasses(Car, Vehicle)
(Car has (superclasses (Vehicle)))

;;; "Cars have four wheels, an engine, and a chassis"
;;; ∀c isa(c, Car) → wheel-count(c, 4)
;;; ∀c isa(c, Car) → ∃e, ch isa(e, Engine) ∧ isa(ch, Chassis) ∧ parts(c, e) ∧ parts(c, ch)
(every Car has
  (wheel-count (4))
  (parts ((a Engine) (a Chassis))))
```

The expression `(a Engine)` denotes that there exists an instance of engine for each car.

Save the text file (eg. as file `demo.km`), then load it into the KM environment using the `load-kb` command:

```
KM> (load-kb "demo.km")
```

We can now issue queries to this (tiny) KB. For example, we can "create" (assert the existence of) a car using the expression `(a Car)`, and ask for the value(s) of that car's slot using a query of the following form:

```
(the <slot> of <expr>)
```

For example:

```
;;; "There is a car."
;;; ∃c isa(c, Car)
KM> (a Car)
(_Car0)
```

;;; “What are the parts of that car?”
;;; $ANS = \{\ x \mid parts(\_Car0, x)\ \}$

```
KM> (the parts of _Car0)
(_Engine2 _Chassis3)
```

where ANS denotes the answer to the query (formally, ANS is a function mapping the query + current state of the KB to an answer, typically a set of instances). As can be seen, queries are always asked and answered in terms of instances. The expression `(a Car)` asserts that $\exists c\, isa(c, Car)$ and, when evaluated, causes a Skolem constant denoting that that car instance to be created. KM creates a frame for that instance, and places it as an instance of the class `Car` in the taxonomy. The KM interpreter thus behaves as follows when answering the query `(the parts of (a Car))`:

1. an instance of car is created (`_Car0`, say)
2. The expressions on the `parts` slot on `_Car0` are found and evaluated. In this case, `_Car0` inherits the expressions `(a Engine) (a Chassis)`, denoting instances of an engine and a chassis. Evaluation of these expressions causes Skolem constants to be generated to denote these instances (eg. `_Engine2`, `_Chassis3`).

More generally, <`expr`> can be any KM expression, for example we can ask:

;;; “What are the parts of a car?”
;;; $\exists c\ isa(c, Car)\ \wedge\ ANS = \{\ x \mid parts(c, x)\ \}$

```
KM> (the parts of (a Car))
(_Engine7 _Chassis8)
```

where again ANS denotes the answer to the query (ie. is a function from query + KB to an answer). In this case, the expression contains both an assertional and query component to it. In future in this manual, we will abbreviate $ANS = \{\ldots\}$ to be simply $\{\ldots\}$ when the query contains no assertional parts, for ease of reading.

3.3 Inheritance

The previous example illustrates *inheritance* in KM, namely the process of an instance acquiring information from the class(es) to which it belongs. In this example, the instance (`_Car0`) inherited information fron a single class (`Car`). More generally, an instance will inherit information from *all* the classes to which it belongs, namely the instance's immediate classes, and (recursively) all those classes' superclasses. (KM supports multiple inheritance, i.e., an instance may be in multiple classes, and a class may have multiple superclasses). When there are multiple, inherited expressions, KM evaluates them and then combines the results together through a process called unification. This is described in Section 10.

3.4 Naming Conventions

To clearly distinguish classes, instances, and slots, it is useful to follow some (optional) naming conventions when writing a KB, such as:

classes	begin with an upper-case letter, eg. `Car`
slots	begin with a lower-case letter, eg. `parts`
instances	begin with a `*` prefix, `*Pete`
anonymous instances	begin with a _ prefix, eg. `_Car21`
variables	begin with a `?` prefix, eg. `?x`, `?car`

Anonymous instances (Skolem individuals) are generated by KM at run-time to denote existentially quantified objects. They differ from named instances in that they are *not* subject to the unique names assumption, ie. they can be unified together (Section 10). We follow these conventions below, though they are (except for the last two) optional. If you'd prefer to use a different convention for instances (eg. use a different prefix, producing `$Pete`, `%Pete`, say), or no convention for visually identifying instances (ie. just `Pete`), then KM will still work fine (KM distinguishes instances from classes by surrounding syntax, eg. `(a Pete)` vs. `Pete`), although it is recommended to distinguish instances in *some* way to remind the KB author whether (say) `Red` was intended to denote an instance or a class, and thus ensure the symbol is used consistently.

To use frame names with other special symbols in (namely most other non-alphanumeric characters, such as a space, comma, parenthesis), the frame name needs to be surrounded with vertical bars, e.g.

```
KM> (|My car| has              ; frame name includes a space character
       (color (*Blue)))
(|My car|)
```

3.5 KM's Printing of Anonymous Instance Names

During reasoning, KM may create anonymous (Skolem) instances, denoting some anonymous member of a class. Although the name of such an instance is arbitrary, KM tries to generate a meaningful name using the class which that instance belongs to. For example, an anonymous instance of the class `Car` may be called `_Car23`. Thus, in general, the name of a Skolem instance gives a clue as to the class it belongs to. Note, though, that there is no *requirement* that the name reflect the class, and in some cases the Skolem name may be misleading as to its immediate class. To reduce confusion, in cases where the instance's name does not reflect its most specific class, or where the instance has multiple most specific classes, KM will print out a comment next to the instance's name listing its class(es). To actually request the name of the instance's classes, use one of the synonymous KM commands:

```
(the classes of instance)
(the instance-of of instance)
```

For example:

```
KM> (_MyCar has
       (instance-of (Jaguar)))
(_MyCar #|Jaguar|#)                 ; class of _MyCar listed in comment

KM> (the classes of _MyCar)
(Jaguar)
```

Note `#|Jaguar|#` is merely a comment in KM's printed output, and not part of the answer itself. The printing of these comments can be disabled by doing at the Lisp or KM prompt:

```
KM> (SETQ *ADD-COMMENTS-TO-NAMES* NIL)
```

3.6 Viewing Frames

To view a frame, the following commands are used:

```
(showme <instance>)          ; display <instance>
(showme (thelast <class>))   ; display the most recent instance of <class>
(showme <class>)             ; display <class>
(showme <expr>)              ; display the instance(s) which <expr> refers to
```

For example:

```
KM> (showme _Car0)         ; refers to _Car0
(_Car0 has
   (instance-of (Car))
   (parts (_Engine1 _Chassis2)))

KM> (showme (thelast Car))    ; refers to the most recent instance of Car (= _Car0)
(_Car0 has
   (instance-of (Car))
   (parts (_Engine1 _Chassis2)))

KM> (showme Car)           ; refers to the class Car
(Car has
  (superclasses (Vehicle))
  (instances (_Car0)))

(every Car has
  (wheel-count (4))
  (parts ((a Engine) (a Chassis))))
```

Note that `_Car0` does not have a known `wheel-count`. KM only computes the values of slots on demand, and because we haven't issued a query asking for the `wheel-count` of `_Car0`, it is thus not recorded on the `_Car0` frame. Asking for this slot value would cause the expression '`4`' to be inherited from the `Car` frame, evaluated ('`4`' evaluates to itself), and the result added to the `_Car0` frame.

While `(showme` *expr*`)` shows just local information on a frame, the command `(showme-all` *expr*`)` shows (but does not evaluate) local *and* inherited information for that frame:

```
KM> (a Car)
(_Car4)

KM> (showme _Car4)
(_Car4 has
  (instance-of (Car)))

KM> (showme-all _Car4)
(_Car4 has
  (instance-of (Car))
  (parts ((a Engine) (a Chassis)))
  (wheel-count (4)))
```

4 Slots

4.1 Overview

Slots denote binary[2] relations between instances, and are themselves treated as instances of the built-in class `Slot`. They do not have to be explicitly declared, unless they require some non-default property (eg. a non-default cardinality), or the user wishes more rigorous knowledge-base checking (Section 6.7 and 6.8).

Slots are declared using the usual frame structure for instances, for example:

---

[2] KM also supports use of N-ary slots, described later in Section 29.1

```
(parts has
  (instance-of (Slot))
  (domain (Physobj))
  (range (Physobj))
  (cardinality (1-to-N))      ; ie. an object can have many parts, but any part is
  (inverse (part-of))         ;       part of at most one object
  (subslots (mechanical-parts body-parts)))

(mechanical-parts has
  (instance-of (Slot))
  (superslots (parts)))       ; mechanical-parts is a subslot of parts

(body-parts has
  (instance-of (Slot))
  (superslots (parts)))       ; body-parts is a subslot of parts

(wheel-count has
  (instance-of (Slot))
  (domain (Vehicle))
  (range (Integer))
  (cardinality (N-to-1)))
```

There are six[3] built-in slots for slots described here:

`domain`	The most general class(es) allowed for the slot's first argument (i.e., the instance using the slot). If multiple classes are specified then they are treated disjunctively, i.e., the instance must be in (at least) one of those classes.
`range`	The most general class(es) allowed for the slot's second argument (i.e., each filler of the slot).
`cardinality`	Given one argument, will there be at most 1 or n values for the other? Must be one of `1-to-1`, `1-to-N`, `N-to-1`, `N-to-N`. Examples of slots with these cardinalities are `spouse`, `is-father-of`, `has-father` and `has-parent` respectively. The default cardinality is `N-to-N`.
`inverse`	The name of the slot if the argument order is reversed. If unspecified, the default inverse name for *slot* is *slot*`-of`.
`subslots`	'more specific' slots, eg. `has-sons` is a subslot of `has-children`
`superslots`	'more general' slots (the inverse of `subslots`)

The cardinalities `1-to-1` and `N-to-1` state there is *at most* one value for that slot. To state there is *exactly* one, the user should also specify a value on the frame using that slot, or use the constraint mechanism (Section 12). For example, suppose we wish to state that a person has exactly one age. We would define the slot `age` as having cardinality `N-to-1` (ie. at most one age), and then also declare that every person has an age:

```
KM> (every Person has
        (age ((a Number))))
```

An inverse slot will necessarily have the reverse cardinality, and switched domain and ranges, of the original slot. Inverse slots do not need to be explicitly declared, as KM will compute their details from the original slots.

---

[3] Plus a few special case slots for slots, described later

Subslots are used for 'slot inheritance': when computing the value(s) of a slot, KM will also look and combine all the values of its subslot(s). This is described later in Section 4.4. Normally only subslots or superslots need be declared, and KM will install the inverse automatically at load time.

For some special purposes, it may be desirable that a slot's values (fillers) are classes rather than instances. The range of these slots will thus be the metaclass `Class`. Section 29.11 discusses this in more detail, and describes how to specify rules for combining these class values together.

4.2 Inverses

KM automatically keeps track of inverse relations, so for example as `_Car0` has parts `_Engine1`, then `_Engine1` must necessarily be part-of `_Car0`. In the absence of other information, KM creates a name for the inverse relation by adding (or removing) the postfix `-of` to the relation name, eg. the inverse of `parts` is `parts-of`:

```
KM> (*Fred has
       (loves (*Sue)))

KM> (showme *Sue)
(*Sue has
  (loves-of (*Fred)))           ; note inverse
```

The naming of inverse slots can also be explicitly declared in a slot declaration as just described. In addition, several of KM's built-in slots do not follow this default `"-of"` convention, including:

Slot	**Inverse slot**	**Purpose**
`instance-of` `classes`	`instances`	instance-class relationship (`instance-of` and `classes` are synonym slots)
`superclasses`	`subclasses`	class-class relationship
`superslots`	`subslots`	slot-slot relationship
`inverse`	`inverse`	slot-slot relationship

Slot inverses are automatically coerced to be in the class `Slot`, in the absence of any other declaration. Also, declaring the domain/range on a slot will result in the opposite range/domain being automatically asserted on that slot's inverse.

4.3 Transitive Closure Slots

Some additional slots are provided to compute the transitive closure of several built-in slots. The results are not cached on the frame, but recomputed each time the slot is queried. These slots are:

Slot	**Computes Transitive Closure Of**
`all-instances`	`instances`
`all-classes`	`classes`, `instance-of` (synonyms)
`all-subclasses`	`subclasses`
`all-superclasses`	`superclasses`
`all-subslots`	`subslots`
`all-superslots`	`superslots`
`all-prototypes`	`prototypes` (Section 22)
`all-supersituations`	`supersituations` (Situations Manual [8])

4.4 Slot Hierarchies

KM allows the user to specify a slot hierarchy using the `subslots` relation. Subslots have the semantics (quantifying over relations *subs* and *s* below):

$$\forall x, y, s, subs \;\; subs(x, y) \wedge \texttt{subslot}(s, subs) \rightarrow s(x, y)$$

Operationally, this means that when computing the value(s) of a slot, KM will also look and combine all the values of its subslot(s) together. For example consider the KB:

```
(Vehicle has (superclasses (Physobj)))

(every Vehicle has
  (body-parts ((a Frame) (a Fender)))
  (parts ((a Steering-wheel))))

(Car has (superclasses (Vehicle)))

(every Car has
  (mechanical-parts ((a Engine)))
  (body-parts ((a Frame))))

;;; SLOT HIERARCHY:
;;;           parts
;;;          /     \
;;; body-parts    mechanical-parts
(parts has
  (instance-of (Slot))
  (subslots (mechanical-parts body-parts)))
```

Thus:

```
KM> (the parts of (a Car))
(_Steering-wheel13 _Engine14 _Frame15 _Fender17)
```

5 Access Paths

5.1 Overview

The earlier query of the form (`the` *slot* `of` *instance*) in Section 3.2 is referred to as a *path*, as, thinking of the KB as a semantic net, it corresponds to a chain of arcs (here of length 1) from *instance* to the answer instances. A query (`the` $slot_2$ `of` (`the` $slot_1$ `of` *instance*)) is a path of length 2. Formally, we define a path as a chain of binary predicates P_i linking some individual $X0$ to other individuals x_n, such that the second argument of P_i is the same as the first argument of P_{i+1}:

$$P_1(X0, x_1), P_2(x_1, x_2), ..., P_n(x_{n-1}, x_n)$$

where the x_i's are free variables. A path denotes the set S of values for x_i (the last variable in the path) for which there exists at least one value for all the other variables in the path, ie.

$$\{ \; x_n \mid \exists x_1, \ldots, x_{n-1} \;\; P_1(X0, x_1) \wedge \ldots \wedge P_n(x_{n-1}, x_n) \; \}$$

(where $\{x|expr(x)\}$ denotes the set S such that $\forall x \; expr(x) \leftrightarrow x \in S$). For example, the access path $parent(John, x), sister(x, y)$ denotes "John's parents' sisters". In KM notation this would be written:

```
(the sister of (the parent of *John))
```

In KM, we add the additional feature that the variables x_n can be typed (`class1`, `class2`, etc.). The initial starting-point for a path is specified by a KM expression <`expr`>, which evaluates to one (or more) instances. The full KM notation for paths is as follows:

`(the` $class_n$ $slot_n$ `of (the` $class_{n-1}$ $slot_{n-1}$ `of (... of (the` $class_1$ $slot_1$ `of` $expr$ `))))`

(and hence to arbitrary length paths). $class_i$ is the type restriction on variable x_i. To denote no restriction, $class_i$ is omitted, i.e.,:

`(the` $slot_n$ `of (the` $slot_{n-1}$ `of (... of (the` $slot_1$ `of` $expr$ `))))`

For example, the KM interpreter will evaluate the path

`(the` $class_2$ $slot_2$ `of (the` $class_1$ $slot_1$ `of` $expr$`))`

as follows:

1. first evaluate *expr* to find instances I
2. find the value(s) of $slot_1$ on the I(s).
3. select only those which are members of $class_1$
4. find the value(s) of $slot_2$ of those selected instances
5. select only those which are members of $class_2$

5.2 Examples

Enter the following KB using the text editor:

;;; "Cars are vehicles"
;;; $superclasses(Car, Vehicle)$
```
(Car has (superclasses (Vehicle)))
```

;;; "Cars have four wheels, use gas, and have an engine and chassis."
;;; $\forall c\ isa(c, Car) \rightarrow wheel\text{-}count(c, 4)$
;;; $\forall c\ isa(c, Car) \rightarrow uses\text{-}fuel\text{-}type(c, *Gas)$
;;; $\forall c\ isa(c, Car) \rightarrow \exists e, ch\ isa(e, Engine) \wedge isa(ch, Chassis) \wedge parts(c, e) \wedge parts(c, ch)$
```
(every Car has
  (wheel-count (4))
  (uses-fuel-type (*Gas))
  (parts ((a Engine) (a Chassis))))
```

;;; "Gas is a fuel-type, costing $1.34 per unit."
;;; $isa(*Gas, Fuel\text{-}Type) \wedge unit\text{-}cost(*Gas, 1.34)$[4]
```
(*Gas has       ; (Note: no 'every', as *Gas is an instance)
  (instance-of (Fuel-Type))
  (unit-cost (1.34)))
```

;;; "Engines are physical objects"
;;; $superclasses(Engine, Physobj)$
```
(Engine has (superclasses (Physobj)))
```

---

[4] (Throughout this manual we use *isa* as a synonym for *instance-of*)

;;; “Every engine is made of metal.”
;;; $\forall e\ isa(e, Engine) \rightarrow material(e, *Metal)$

```
(every Engine has
  (material (*Metal)))
```

;;; “Chassis’ are physical objects”
;;; $superclasses(Chassis, Physobj)$

```
(Chassis has (superclasses (Physobj)))
```

;;; “Every chassis is made of metal, plastic, and wood.”
;;; $\forall c\ isa(c, Chassis) \rightarrow material(c, *Metal) \wedge material(c, *Plastic) \wedge material(c, *Wood)$

```
(every Chassis has
  (material (*Metal *Plastic *Wood)))
```

;;; “Plastic is a synthetic material.”
;;; $isa(*Plastic, Synthetic\text{-}Material)$[5]

```
(*Plastic has
  (instance-of (Synthetic-Material)))
```

Now reload the KB into the KM environment:

```
KM> (load-kb "demo.km")
```

(`load-kb` can be issued from both the Lisp and `KM>` prompts). and try these examples:

```
KM> (a Car)
(_Car9)
```

;;; “The unit cost of that car’s fuel-type?”
;;; $\{\ c \mid \exists f\ uses\text{-}fuel\text{-}type(\_Car9, f) \wedge unit\text{-}cost(f, c)\ \}$

```
KM> (the unit-cost of (the uses-fuel-type of _Car9))
(1.34)
```

;;; “What are the car’s parts made of?”
;;; $\{\ m \mid \exists p\ parts(\_Car9, p) \wedge material(p, m)\ \}$

```
KM> (the material of (the parts of _Car9))
(*Metal *Plastic *Wood)
```

;;; “What is the engine of that car made of?”
;;; $\{\ m \mid \exists p\ parts(\_Car9, p) \wedge isa(p, Engine) \wedge material(p, m)\ \}$

```
KM> (the material of (the Engine parts of _Car9))
(*Metal)
```

;;; “What is the chassis made of?”
;;; $\{\ m \mid \exists p\ parts(\_Car9, p) \wedge isa(p, Chassis) \wedge material(p, m)\ \}$

```
KM> (the material of (the Chassis parts of _Car9)))
(*Metal *Plastic *Wood)
```

;;; “What are the synthetic materials in that car’s chassis?”
;;; $\{\ m \mid \exists p\ parts(\_Car9, p) \wedge isa(p, Chassis) \wedge material(p, m) \wedge isa(m, Synthetic\text{-}Material)\ \}$

```
KM> (the Synthetic-Material material of (the Chassis parts of _Car9))
(*Plastic)
```

---

[5] (Throughout this manual we use *isa* as a synonym for *instance-of*)

Note that, in the above, evaluating (‘following’) a path may involve visiting multiple frames in the KB.

5.3 Embedded Units and Self

The expression `(a` *class*`)` has a more general form `(a` *class[*`with` *slotsvals]*`)`, allowing additional facts to be asserted about the described instance, as illustrated below:

```
;;; “Create a red car.”
;;; ∃c isa(c, Car) ∧ color(c, ∗Red)
KM> (a Car with (color (*Red)))                  ; a red car
(_Car20)

;;; “What color is it?”
;;; { c | color(_Car20, c) }
KM> (the color of (thelast Car))                 ; what color is it?
(*Red)
```

We can also place such expressions as the value(s) of slots in the KB itself. When placed in a slot-value in the KB, we refer to the expressions as *embedded units*, as they are essentially frame “units” embedded within a larger frame unit in the KB.

Add the following two frames to the demo KB:

```
;;; “Every person has a favorite color.”
;;; ∀p isa(p, Person) → ∃c isa(c, Color) ∧ favorite-color(p, c)
(every Person has
  (favorite-color ((a Color))))

;;; ”Professors have (at least) one car which is old, is their favorite color, and made in USA.”
;;; ∀p isa(p, Professor) → ∃c isa(c, Car)
;;;      ∧ age(c, ∗Old)
;;;      ∧ (∀r favorite-color(p, r) → color(c, r))
;;;      ∧ ∃m isa(m, Manufacturer) ∧ made-by(c, m) ∧ location(m, ∗USA)
(Professor has (superclasses (Person)))

(every Professor has
  (owned-vehicle ((a Car with
                    (age (*Old))
                    (color ((the favorite-color of Self)))
                    (made-by ((a Manufacturer with
                                (location (*USA)))))))))
```

Note the embedded unit `Car` in the `Professor` frame, which itself has an embedded unit `Manufacturer`. Embedded units can be nested arbitrarily deep. A query for the `owned-vehicle` slot of a professor causes this ‘embedded unit’ expression to be evaluated, and a Skolem instance to be created to denote it. That instance will have the stated properties attached to it.

Note also the use of ‘`Self`’ in the above example. `Self` is a keyword, referring to the instance of the outermost frame in which the expression is being evaluated. When the expression is inherited by an instance, KM replaces the `Self` keyword by the name of that instance.

```
;;; “There’s a professor whose favorite color is blue.”
;;; ∃p isa(p, Professor) ∧ favorite-color(p, ∗Blue)
```

```
KM> (a Professor with (favorite-color (*Blue)))
(_Professor25)
```

;;; "Which vehicle(s) does that professor own?"
;;; $\{\ c \mid \textit{owned-vehicle}(\_Professor25, c)\ \}$

```
KM> (the owned-vehicle of (thelast Professor))
(_Car26)

KM> (showme _Car26)
(_Car26 has
   (instance-of (Car))
   (owned-vehicle-of (_Professor25))
   (age (*Old))
   (color ((the favorite-color of _Professor25)))
   (made-by ((a Manufacturer with (location (*USA))))))
```

Note in this instance `_Car26`, created from the embedded unit:

- A copy of the slot-value expressions from the embedded unit have been attached to the instance. Note also that they are *not* evaluated until a query explicitly requests their contents.
- During the copy, the keyword `Self` in the embedded unit has been replaced by the instance of the professor which inherited the information (`_Professor25`). Thus the path in the embedded unit `(the favorite-color of Self)` has become `(the favorite-color of _Professor25)`. Querying for this color causes this expression to be evaluated.

  ;;; $\{\ r \mid \exists c\ \textit{owned-vehicle}(\_Professor25, c) \wedge \textit{color}(c, r)\ \}$

  ```
  KM> (the color of (the owned-vehicle of (thelast Professor)))
  (*Blue)
  ```

  ;;; "Where is the manufacturer of professors' vehicles' located?"
  ;;; $\exists p\ isa(p, Professor) \wedge ANS = \{\ l \mid \exists c, m\ \textit{owned-vehicle}(p, c) \wedge \textit{made-by}(c, m) \wedge \textit{location}(m, l)\ \}$

  ```
  KM> (the location of (the made-by of (the owned-vehicle of (a Professor))))
  (*USA)
  ```

Slots in paths may themselves be expressions (which should evaluate to a single slot).

5.4 Selecting a Subset of Values

Sometimes the user may want to select just a subset of the values which a path points to. The *class* parameter in paths allows a limited form of selection, namely selection of just those values in *class* as described earlier. For finer control, the user can use an `(allof` *path* `where` *test*`)` expression, described later in Section 8, or select particular values by tagging them with identifiers (the 'called' tags), also described later in Section 20.

6 Tracing and Debugging

6.1 Comments

During inference, KM may print comments out to the console, describing what it is doing. These are prefixed by the string `COMMENT:`. Printing of these comments can be disabled/re-enabled by the commands `(nocomments)` and `(comments)` respectively.

6.2 Viewing the KB

The content of frames can be viewed with the `(showme ...)` command, as described earlier in Section 3.6. In addition, the entire knowledge-base (including run-time-generated instances) can be displayed using the command `(write-kb)`, or written to a file using the command `(save-kb` *filename*`)`.

In unusual circumstances, `showme` may show a slot as having duplicate elements in its value set. This is an implementation artifact, due to the fact that KM syntactically removes duplicates "lazily", just when that slot is queried, and does not mean the slot is filled by a bag (rather than set) of values. A query for `(the` *slot* `of` *frame*`)` will cause the duplicates to be removed and the answer returned not contain duplicates, and thus any such syntactic duplicates are invisible to the inference engine.

The command `(print` *expr*`)` is equivalent to *expr*, but will print out the evaluated result as a side-effect. Wrapping expressions in `print` statements may be useful for debugging a knowledge-base, if the normal tracing operations below are not sufficient.

During inference, KM may unify instances together. To see the the list of which instances are unified (bound) to which other instances, type the command `(show-bindings)`.

6.3 The Tracer (Debugger)

To watch KM evaluating a path (or any other expression), the tracer can be turned on by:

```
KM> (trace)
```

and turned off again by:

```
KM> (untrace)
```

Note that these must be issued at the `KM>` prompt, not the Lisp prompt, or it will be interpreted as turn Lisp tracing on/off. To switch the tracer on/off at the Lisp prompt, use the commands `(tracekm)`/`(untracekm)` instead.

The tracer provides a Prolog-like execution trace, and is invaluable for understanding and debugging inference with a KB. By hitting carriage return `<CR>`, you can step through the trace. For example, tracing evaluation of the earlier path in Section 5.2 looks:

```
KM> (trace)

KM> (the Synthetic-Material material of (the Chassis parts of (a Car)))
1 -> (the Synthetic-Material material of (the Chassis parts of (a Car)))
2  -> (the material of (the Chassis parts of (a Car)))
3   -> (the Chassis parts of (a Car))
4    -> (the parts of (a Car))
5     -> (a Car)
5     <- (_Car4208)           "(a Car)"
5     -> (the parts of _Car4208)
5      (1) From inheritance: (:set (a Engine) (a Chassis))
6      -> (:set (a Engine) (a Chassis))
7       -> (a Engine)
7       <- (_Engine4209)      "(a Engine)"
7       -> (a Chassis)
7       <- (_Chassis4210)     "(a Chassis)"
6      <- (_Engine4209 _Chassis4210) "(:set (a Engine) (a Chassis))"
5     <- (_Engine4209 _Chassis4210) "(the parts of _Car4208)"
```

```
4    <- (_Engine4209 _Chassis4210) "(the parts of (a Car))"
3   <- (_Chassis4210)         "(the Chassis parts of (a Car))"
3   -> (the material of _Chassis4210)
3    (1) From inheritance: (:set *Metal *Plastic *Wood)
4    -> (:set *Metal *Plastic *Wood)
4    <- (*Metal *Plastic *Wood) "(:set *Metal *Plastic *Wood)"
3   <- (*Metal *Plastic *Wood) "(the material of _Chassis4210)"
2  <- (*Metal *Plastic *Wood) "(the material of (the Chassis..."
1 <- (*Plastic)               "(the Synthetic-Material mater..."
(*Plastic)
(11 inferences and 118 kb-accesses in 0.0 sec [1100 lips, 11800 kaps])

KM> (untrace)
```

Note above you can see the sub-queries, and the answers to those sub-queries, computed during evaluation of the initial query.

To see the tracing options, type `?` during the trace. These options and their meaning are shown in Table 1. In particular, note that you can skip (`s`) directly to the result of a query, retry (`r`) a query that produces an unexpected result, and ask for more detail during specific operations such as unification (`+U`) and constraint checking (`+C`).

KM also reports the work done in answering a query, in terms of “logical inferences” (recursive calls to the query interpreter) and “knowledge-base accesses” (low-level accesses of the basic frame data structure), both in absolute terms and speed. `lips` stands for “logical inferences per second”, and kaps “knowledge-base accesses per second.”. During reasoning, KM frequently accesses frames to search for constraints, check slot cardinalities, check for subslots, etc., which is why the raw `kaps` figure is often high.

Internally, once KM has computed an instance’s slot’s values while answering a query, those values are asserted locally on that instance’s frame. For efficiency, KM also flags that instance’s slot to note its values have now been computed, and so further calls for those slot-values will simply retrieve them, and *not* cause their recomputation during processing of the rest of the query. Thus in the trace of execution, the first call for a frame’s slot-value will show the computation of those value(s) (eg. through inheritance), whereas subsequent calls will simply show the cached value(s) being retrieved. After the top-level query from the user is finally answered, all ‘already computed’ flags are removed (but the values themselves remain). Thus if the user then repeats the top-level query, the computation will also be repeated.

6.4 Spy Points

KM allows the user to set *spy points* (similar to Prolog), which, when reached, switch on the tracer. A spy point is either a specific KM expression, or an expression pattern. An expression pattern is a KM expression with wild-cards (variable-like symbols) in. Spy points are matched against the current expression being evaluated, and if matched, the tracer is turned on.

For expression patterns, a symbol beginning with a ‘`?`’, e.g., `?x` matches a single element in a list, while the keyword ‘`&rest`’ matches the remainder of a list. Note that these pseudo-variables are not bound, they are simply place-holders.

```
;;; Place a spypoint on (a Car). This will cause KM to automatically
;;; switch on the tracer when the expression (a Car) is evaluated.
KM> (spy (a Car))
```

Option	**Meaning**	**Explanation**
`<cr>,c`	creep	- single step forward
`s`	skip	- jump to completion of current subgoal
`S`	big skip	- jump to completion of parent subgoal
`r`	retry	- redo the current subgoal
`n`	nonstop	- switch off trace for remainder of this query
`a`	abort	- return directly to top-level prompt
`o`	trace off	- permenantly switch off trace
`A`	abort and off	- Equivalent to `a` and `o` together
`f`	fail	- return NIL for current goal (use with caution!)
`z`	zip	- complete query with noninterative trace
`g`	goal stack	- print goal stack
`w`	where	- show which frame the current expression originated from
`d <f>`	display `<f>`	- display (showme) frame `<f>`
`h,?`	help	- this message

Also to show additional detail (normally not shown) for the current query *only*:

`+S`	show additional detail in other situation(s)
`+U`	show additional detail during unification
`+C`	show additional detail during constraint checking
`+X`	show additional detail during classification
`+A`	show absolutely everything (all additional detail)

`-S,-U,-C,-X,-A` to unshow

Table 1: Tracing options and their meanings in KM.

```
;;; Switch on debugger when getting any slot of _Car3
KM> (spy (the ?x of _Car3))

;;; Switch on debugger when getting the parts of any object
KM> (spy (the parts of ?x))

;;; Switch on debugger when evaluating any forall expression
KM> (spy (forall &rest))

;;; Also...
KM> (spy)                          ; list all spypoints

KM> (unspy)                        ; remove all spypoints
```

6.5 Viewing the Taxonomy

The `(taxonomy)` directive prints the taxonomy out, by default descending the `subclasses` and `instances` relationships. Anonymous instances (prefixed with a _) are not included. If the same subtree occurs more than once in the taxonomy (because its has multiple parents), then the first time the entire subtree is displayed, and subsequent times just the characters '`...`' are printed. Instances are preceded with an `I` character to distinguish them from classes. If KM cannot work out if an object is an instance or a class, then it is preceded with a `?` character. Instances are listed first, then subclasses, both in alphabetical order. Only built-in classes which have some information attached to them are included in the taxonomy.

```
KM> (taxonomy)
Thing
   Physobj
      Chassis
      Engine
   Synthetic-Material
I     *Plastic
   Vehicle
      Car
```

In fact, the current KM supports a generalized version of this command:

```
KM> (taxonomy <concept-to-start-at> <relation-to-descend>)
```

where `<concept-to-start-at>` defaults to `Thing`, and
`<relation-to-descend>` defaults to `subclasses`
(also showing `instances`) as described above

For example:

```
KM> (taxonomy)                     ; show subclass tree from Thing down
KM> (taxonomy Physobj)             ; show subclass tree from Physobj down
KM> (taxonomy _Car1 direct-parts)  ; show partonomy of _Car1
```

6.6 Behavior on Failure

By default, if a top-level query returns no values, KM simply prints out `NIL`:

```
KM> (the foo of bar)
NIL
```

However, this behavior can be changed so KM prints out an error instead by the command `(fail-noisily)` (and conversely with `(fail-quietly)`):

```
KM> (fail-noisily)
KM> (the foo of bar)
ERROR! No values found for (the foo of bar)!
Switching on debugger...
1 <- FAIL!                                [(the foo of bar)]
```

This is particularly useful when creating a test file for a KB. The test file contains KM queries, all of which are expected to succeed. To ensure the user is informed if any fail, add the command `(fail-noisily)` at the start of the file. Then, to test all queries succeed, simply load the test file (using `(load-kb "`*file*`")`) and ensure it loads without error.

For additional control of KM's behavior on failure and on error, see Section 26.2.

6.7 Load-Time Knowledge-Base Checking

When loading a KM KB, only limited syntax checking is performed. In particular, KM does not currently have its own parser, and instead performs only limited checking of the entered expression before storing it. Additional checking is performed at run-time (Section 6.8).

For additional load-time KB checking, the command `(scan-kb)` will list all symbols in the KB which do not have a frame defined for them (eg. are mentioned on a slot's value, but are not defined). Note that these are warnings, not necessarily errors, as it is not necessary to explicitly declare all the frames used (for example, slots do not need to be explicitly declared unless some non-default cardinality, inverse name, or other features is required). However, this function can be useful for debugging a knowledge base and catching typing errors.

6.8 Query-Time Knowledge-Base Checking

During inference, KM can be made to perform certain checks on the instances/classes/slots it uses. There are two kinds of checks which can be made:

- Checking that all given classes and slots have frames explicitly declaring them in the KB (`(checkkbon)`/`(checkkboff)`).

  Although KM does not require that all mentioned symbols have a frame, it is sometimes desirable to strive for this to make sure that important information is not accidentally omitted from the KB.

- Checking that the domain and range constraints of a slot are not violated by any any computation (`(enable-slot-checking)`/`(disable-slot-checking)`).

  Note that this does not check that all slot-values are instances of a slot's range, merely that they are not inconsistent with a slot's range. For example, if the domain of the slot `spouse` is `Person`, the value `_Thing1`, an instance of the class `Thing`, would *not* be considered an error (as `_Thing1` may later turn out to be a person).

To switch the former on, do:

```
> (checkkbon)
```

And to turn it off, do:

```
> (checkkboff)
```

This command can be issued from the Lisp or KM prompt, or included in a knowledge-base file. Domain and range checking can be switched on/off in a similar way.

It should be noted that declaring and checking slots in these ways is not essential for inference (though may help reduce errors during KB development), and thus by default the checkers are turned off. Also note that these only do *run-time* checking, ie. during inference, and do not perform any extra load-time verification of the KB itself. These facilities augment the normal tracing/debugging facility described in Section 6.

In the examples in this manual, the (tiny) knowledge bases contain enough for inferencing, but do not include all the slot declarations and inheritance information that would normally be associated with the KB. If you wish to create the full KB, then turn KB checking on during inference to receive pointers to where declarations are missing.

In addition, KM's keywords (`a`, `the`, `forall`, etc.) cannot be used as frame names. KM will report an error if such an attempt is made – these error messages are particularly helpful in debugging misplaced parentheses in KM expressions. A full list of these keywords is given in the BNF for KM in the Reference Manual [10].

7 Conditionals

7.1 Conditional Expressions

An access path can be viewed as a simple form of "rule" when placed on a slot, describing how to compute that slot's value (ie. the relation's second argument, given its first). In fact, there is a wide variety of rule expressions that a user may need to write and which KM supports, which we now describe in this and subsequent sections.

KM supports conditional `if...then...` expressions, where the value returned depends on some test(s). The forms of these are as follows:

`(if` *expr* `then` *expr [* `else` *expr]*`)`
`(`*expr* `and` *expr [*`and` *expr]**`)`
`(`*expr* `or` *expr [*`or` *expr]**`)`
`(not` *expr*`)`
`(`*expr* `=` *expr*`)`
`(`*expr* `/=` *expr*`)`
`(`*expr* > *expr*`)`
`(`*expr* < *expr*`)`
`(`*expr* >= *expr*`)`
`(`*expr* <= *expr*`)`
`(`*expr* `includes` *expr*`)`
`(`*expr* `is-superset-of` *expr*`)`
`(`*expr* `isa` *class*`)`
`(has-value` *expr*`)`
`(numberp` *expr*`)`

In `if...then...` expressions, the condition is deemed to be "true" if it returns a non-nil answer, in which case the consequent *expr* is evaluated and the result returned. For comparison operations, eg. `=`, KM returns the special symbol `t` to denote success.

For `and` expressions, KM evaluates each *expr* in turn until either one returns no values, or all are evaluated (in which case the result of the last evaluation is returned). For `or` expressions, KM evaluates each *expr* in turn until one expression returns some values.

KM treats `not` as negation-by-failure, ie. the failure to find any values for *expr* means that `(not` *expr*`)` is true. `X /= Y` is interpreted as `(not (X = Y))`. >, >=, <, and <= are arithmetic tests, whose expressions must evaluate to (a singleton set containing) a number (KM will coerce those singleton sets to their contained number before making the test). `isa` checks whether an instance is a member of a class, and again its arguments must both evaluate to (sets containing a) single instance/class (otherwise an error is reported).

`=` is used for testing both equality of instances and sets, and will coerce an instance to be a set if necessary for type compatibility:

```
KM> ((:set 1 2) = (:set 2 1))
(t)

KM> ((1 + 1) = 2)
(t)
```

(Note we use `:set` to declare the expression is a set of KM expressions, not a single KM expression). The operator `includes` tests set membership, and requires its second argument to be an instance or a singleton set (which will be coerced to be an instance). `is-superset-of` performs set comparison:

```
KM> ((:set 1 2 3) includes 1)
(t)

KM> ((:set 1 2 3) is-superset-of (:set 1 2))
(t)
```

The below illustrates a conditional as a slot's value:

;;; ”If a car is domestic then spare parts will be cheap, otherwise they'll be expensive.”
;;; $\forall c\ isa(c, Car) \rightarrow$
;;; $(\ (\ \exists l, l', m, o\ made\text{-}by(c,m) \wedge location(m,l) \wedge\ owner(c,o) \wedge lives\text{-}in(o,l') \wedge\ l = l'\ )$[6]
;;; $\rightarrow cost\text{-}of\text{-}parts(c, *Low)\ ;\ cost\text{-}of\text{-}parts(c, *High)\ )$[7]

```
(every Car has
  (cost-of-parts (
       (if    ((the location of (the  made-by of Self)) =
               (the lives-in of (the owner of Self)))
        then  *Low
        else  *High))))

(Sentra has (superclasses (Car)))              ; Sentras are a type of Car...

(every Sentra has (made-by (*Nissan)))         ; made by Nissan.

(Geo-Metro has (superclasses (Car)))           ; Geo-Metros are a type of Car...

(every Geo-metro has (made-by (*Chrysler)))    ; made by Chrysler.

(*Nissan has                                   ; Nissan is a Japanese manufacturer
  (instance-of (Manufacturer))
  (location (*Japan)))

(*Chrysler has                                 ; Chrysler is an American manufacturer
  (instance-of (Manufacturer))
  (location (*USA)))
```

Note that the value of the car's `cost-of-parts` slot is a conditional expression, which tests whether the car's manufacturer's country equals the country that the owner lives in. If so, then the value `*Low` is returned (ie. domestic cars have low-cost parts), otherwise the value `*High` is returned. Thus for a Japanese car owned by an American, the cost of spare parts will be high:

---

[6] This semantics assumes manufacturers and people have only one location. More literally, the KM expression represents “domestic” as “the set of manufacturer locations = the set of owner locations”, ie. $\{\ l \mid \exists m\ made\text{-}by(c,m) \wedge location(m,l)\ \} = \{\ l' \mid \exists o\ owner(c,o) \wedge lives\text{-}in(o,l')\ \}$

[7] $(X \rightarrow Y\ ;\ Z)$ is a shorthand notation for $(X \rightarrow Y) \wedge not(X) \rightarrow Z$, where $not()$ is negation by failure.

```
;;; “Create a Sentra (Japanese car) owned by a person living in USA.”
;;; ∃c, p isa(c, Sentra) ∧ isa(p, Person) ∧ owner(c, p) ∧ lives-in(p, *USA)
KM> (a Sentra with
       (owner ((a Person with (lives-in (*USA))))))
(_Sentra75)

;;; “How much are its spare parts?”
;;; { c | cost-of-parts(_Sentra75, c) }
KM> (the cost-of-parts of _Sentra75)
(*High)
```

7.2 Conditionals and the Closed-World Assumption

When applying tests, KM makes a **closed-world assumption**, i.e., if an expression evaluates to nil, KM assumes that the expression denotes zero objects (rather than accommodating the possibility that some of the values might be unknown, due to missing information in the KB). Similarly, if an *expr* evaluates to nil, KM assumes `(not` *expr*`)` is true (negation as failure). This should be born in mind by the knowledge engineer.

If information may be unknown, then the knowledge engineer should explicitly test for this using the `has-value` or `numberp` tests to avoid mistaken conclusions by the closed-world assumption. `(has-value` *expr*`)` tests whether an expression has a known value, and implementationally is equivalent to *expr*. `(numberp` *expr*`)` test whether an expression evaluates to a specific numeric value (as opposed to, say, `_Number23`). An example of their use is:

```
;;; “Foreign-made cars have expensive parts.”
;;; “If the car is more than 4 years old, then it’s old.”
KM> (every Car has
      (cost-of-parts (
        (if   (     (has-value (the location of (the made-by of Self)))
                and ((the location of (the  made-by of Self)) =
                     (the lives-in of (the owner of Self))))
         then  *Low else  *High)))
      (is-old  ((if   (numberp (the age of Self))
                   then (if    ((the age of Self) < 5)
                         then *No else *Yes)))))
```

The use of `numberp` and `has-value` prevents these rules firing, and hence making assumptions about, cars of unknown age or manufacturer location.

8 Universal Quantification

In addition to class-level `(every ... has ...)` declarations, KM allows universal quantification to be used in slot-value expressions. These allow the user to select from, manipulate, and test members of sets. KM has five operators for this purpose, as follows:

`(allof` *[var* in*] expr1* `where` *expr0*`)`
`(forall` *[var* in*] expr1 [*`where` *expr0] expr2*`)`
`(oneof` *[var* in*] expr1* `where` *expr0*`)`
`(theoneof` *[var* in*] expr1* `where` *expr0*`)`
`(allof` *[var* in*] expr1 [*`where` *expr0]* `must` *expr2*`)`

It is important to note that these keywords perform tests, rather than make assertions. `allof` sieves out members of a set *expr1* which pass a test *expr0*, while `forall` sieves out these members then returns some computation *expr2* applied to each member of that set. There is thus both a procedural as well as declarative aspect to such expressions.

In addition, the operator `oneof` behaves like `allof`, except it returns just the *first* item passing a test. `theoneof` performs an additional test that there is, in fact, exactly one such item, and will report an error if otherwise.

Finally, `allof...must...` tests that *all* members of *expr1* where *expr0* holds, also pass the test *expr2*. If the test is passed, the expression returns the special instance `t`, denoting true.

To refer to the "current member being tested" within *expr0* and *expr2*, the user has two choices:

1. If the optional "*var* `in`" construct is used, the user provides a variable name (a symbol beginning with a '`?`' character), which *expr0* and *expr2* can refer to.
2. If the optional "*var* `in`" construct is not used, the special keyword `It` refers to the item being tested.

Here are some examples to illustrate their use:

```
;;; "Joe owns an old brown car, a new red car, and an old red car."
KM>  (*Joe has
        (instance-of (Person))
        (owns ((a Car with (color (*Brown)) (age (*Old)))
               (a Car with (color (*Red))   (age (*New)))
               (a Van with (color (*Red))   (age (*Old))))))

;;; "Which red things does Joe own?"
;;; { x | owns(*Joe, x) ∧ color(x, *Red) }[8]
KM> (allof (the owns of *Joe)
     where ((the color of It) = *Red))
(_Car21 _Van22)

;;; Or equivalently...
KM> (allof ?x in (the owns of *Joe)
     where ((the color of ?x) = *Red))
(_Car21 _Van22)

;;; "Show me one of Joe's red things."
;;; oneof({ x | owns(*Joe, x) ∧ color(x, *Red) })[9]
KM> (oneof (the owns of *Joe)
     where ((the color of It) = *Red))
(_Car21)

;;; Or equivalently...
KM> (oneof ?car in (the owns of *Joe)
     where ((the color of ?car) = *Red))
(_Car21)

;;; "What are the age(s) of Joe's red things?"
;;; { a | ∃x owns(*Joe, x) ∧ color(x, *Red) ∧ age(x, a) }
KM> (forall (the owns of *Joe)
      where ((the color of It) = *Red)
```

---

[8] This semantics assumes an object has exactly one color. The more general semantics for `((the color of` x`) = *Red)` is $\{ c \mid color(x,c) \} = \{*Red\}$ ("x's colors are only red"), which reduces to the former under this assumption.

[9] where *oneof*() is a deterministic but unspecified function mapping a set onto one of its members.

```
            (the age of It))
(*New *Old)

;;; Or equivalently...
KM> (forall ?x in (the owns of *Joe)
      where ((the color of ?x) = *Red)
            (the age of ?x))
(*New *Old)

;;; "Which are Joe's vans?"
;;; { v | owns(*Joe, v) ∧ isa(v, Van) }
KM> (allof (the owns of *Joe)
     where (It isa Van))
(_Van22)

;;; "Which are Joe's vans?" (alternative form of same query, same semantics)
KM> (the Van owns of *Joe)
(_Van22)

;;; "Which are Joe's new cars?"
;;; { c | owns(*Joe, c) ∧ isa(c, Car) ∧ age(c, *New) }
KM> (allof (the owns of *Joe)
     where ((It isa Car) and ((the age of It) = *New)))
(_Car21)

;;; "What are the color(s) of Joe's new car(s)?"
;;; { r | ∃c owns(*Joe, c) ∧ isa(c, Car) ∧ age(c, *New) ∧ color(c, r) }
KM> (the color of (allof (the owns of *Joe)
                   where ((It isa Car) and ((the age of It) = *New))))
(*Red)

;;; "Red is a pretty color."
KM> (*Red has (instance-of (Pretty-color)))
(*Red)

;;; "What are all Joe's pretty-colored things?"
;;; { x | owns(*Joe, x) ∧ color(x, c) ∧ isa(c, Pretty-color) }[10]
KM> (allof (the owns of *Joe)
     where ((the color of It) isa Pretty-color))
(_Car21 _Van22)

;;; "Are all Joe's vans red?"
;;; ( (∀v v ∈ { x | owns(*Joe, x) ∧ isa(x, Van) } → color(v, *Red) ) → ANS = {t}
KM> (allof (the owns of *Joe)
     where (It isa Van)
     must  ((the color of It) = *Red))
(t)
```

Note, of course, that these expressions can be placed on slots in a frame, as well as issued at the top-level prompt. For example:

```
;;; [1] below: "A Car's engine turns the front wheels."
```

---

[10] KM's `isa` test assumes both its arguments have exactly one value (and will generate an error otherwise), see Section 7.

```
;;; ∀c isa(c, Car) → ∃e isa(e, Engine) ∧ has-engine(c, e)∧
;;;      ( ∀w parts(c, w) ∧ isa(w, Wheel) ∧ position(w, *Front) → powers-wheels(e, w) )
(every Car has
  (parts ((a Wheel with (position (*Front)) (side (*Left)))
          (a Wheel with (position (*Front)) (side (*Right)))
          (a Wheel with (position (*Back)) (side (*Left)))
          (a Wheel with (position (*Back)) (side (*Right)))
          (a Chassis)
          (Self has-engine)))
  (has-engine ((a Engine with                                   ; [1]
                 (powers-wheels ((allof (the Wheel parts of Self)
                                  where ((the position of It) = *Front))))))))
```

The expression `(allof (the Wheel parts of Self) where ((the position of It) = *Front))` refers to the front wheels of the car. When evaluating it, the KM interpreter first finds all the wheel parts of the car (evaluates `(the Wheel parts of Self)`), and then iterates through each of those (four) wheels, selecting just those which satisfy the test `((the position of It) = *Front)` (ie. just the front wheels), where It refers to the instance being tested. Thus:

```
KM> (*MyCar has (instance-of (Car)))

;;; "What wheels are on my car?"
;;; { w | parts(*MyCar, w) ∧ isa(w, Wheel) }
KM> (the Wheel parts of *MyCar)
(_Wheel121 _Wheel122 _Wheel123 _Wheel124)

;;; "Which wheels do the engines power?"
;;; { w | ∃e parts(*MyCar, e) ∧ isa(e, Engine) ∧ powers-wheels(e, w) }
KM> (the powers-wheels of (the Engine parts of *MyCar))
(_Wheel121 _Wheel122)

;;; Check these two wheels are the front wheels!
KM> (the position of _Wheel121)
(*Front)

KM> (the position of _Wheel122)
(*Front)
```

`forall` (etc.) statements can be nested (in which case explicit variable names for the iterator should be used, to avoid ambiguity). For example:

```
;;; "Joe owns a new, green car."
;;; ∃c isa(c, Car) ∧ owns(*Joe, c) ∧ color(c, *Green) ∧ age(c, *New)
KM> (*Joe has
      (instance-of (Person))
      (owns ((a Car with (color (*Green)) (age (*New))))))

;;; "Show me all the colors of all the things owned by all the people."
;;; { c | ∃p, o isa(p, Person) ∧ owns(p, o) ∧ color(o, c) }
KM> (forall ?person in (the all-instances of Person)
           (forall ?owned in (the owns of ?person)
                   (the color of ?owned)))
(*Green *Brown *Red)
```

```
;;; (The above query can also equivalently written as below)
KM> (the color of (the owns of (the instances of Person)))
(*Green *Brown *Red)
```

Historical Note: The additional operators `forall2`, `allof2`, `oneof2`, and their corresponding keyword `It2`, used in prior releases in KM, are supported but considered obsolete now in KM.

9 Local Variables

The previous Section illustrated an optional use of variables in `forall`, `allof`, and `oneof` expressions. KM also allows variables to be used in one other way, namely within conjunctive ('`and`') expressions. The general form is:

`((`*var* `==` *expr*`)` *[* `and` *expr]+*`)`

where *var* is a symbol beginning with a '`?`', and the scope of the variable is *within the rest of the conjunctive expression* (only). For example, the expression:

```
KM> (every Person has
      (nationality ((if ((the lives-in of Self) = *America) then *American))))
```

could be expressed equivalently:

```
KM> (every Person has
      (nationality ((     (?x == (the lives-in of Self))
                     and (if (?x = *America) then *American)))))
```

This local variable mechanism is purely a syntactic convenience, useful in large expressions which use the same path multiple times.

As a separate tip, note that instances can be used somewhat like global constants, where an "assignment" is performed using KM's unification operator `==` or its synonym `&` (see next Section for details on unification):

```
KM> (X1 == (a Car with (color (*Red))))      ; == is a synonym for &

KM> (the color of X1)
(*Red)

KM> (showme X1)
(X1 has
  (color (*Red))
  (instance-of (Car)))
```

This thus provides a syntactically convenient way of assigning a name to a frame (equivalent to writing `(X1 has (instance-of (Car)) (color (*Red)))`). Note that `X1` is in fact an instance frame (not a variable), and `==` is doing unification (not assignment).

10 Unification, Equality, and Multiple Inheritance

10.1 Introduction

An instance may inherit information from multiple generalizations, either 'vertically' in the inheritance (superclass) hierarchy (eg. `_Car1` inherits from `Car`, `Vehicle`, and `Physobj`), 'horizontally' (eg. `_Pet-Fish01` inherits from `Pet` and `Fish`), or both. The semantics of KM dictate that the instance will acquire all the properties of all its generalizations (ie. all the axioms applying to its generalizations also apply to the instance). This requires that, when computing the value of an instance's slot, KM will search and merge information from *all* that instance's generalizations. This merging process is called *unification*. Syntactically, unification is the process of combining the data structures representing different instances. Semantically, unification is the assertion of *equality* between those instances. The syntactic operation implements the semantic operation, because the syntactically merged data structure represents an object which has all the properties of all the unified instances.

Unification is one of the important differentiators between logic-based frame languages like KM, and object-oriented programming languages. In an object-oriented language, methods from different classes are not automatically 'merged' together, but instead one will take precedence over another. For example, in Smalltalk (a single-inheritance language), a method on an instance's most specific class overrides (rather than is merged with) definitions for the same method on other, more general classes. Similarly, in C++ (a multiple-inheritance language), a reference to a method defined in multiple parents must be disambiguated so that it is clear which one is to be used (rather than the compiler somehow 'merging' and using the multiple methods together). (See [11] Chapter 12 for further discussion).

Using unification to combine multiple pieces of inherited information is fundamental for building KBs in a modular, reusable fashion. Ideally, a concept's representation will be 'layered', meaning that the more general (and reusable) axioms are separated from the more idiosyncratic, concept-specific axioms. Thus rather than placing all relevant information about a concept on that concept's frame, it will be distributed among that concept's generalizations, with the more general (and reusable) axioms placed on concepts high up in the taxonomy. This separation of knowledge provides the basis for reuse, as the more general axioms can now be inherited and applied to other concepts as well as the original one.

10.2 Value Unification, Single-Valued Slots, and the Unification Operator `&`

10.2.1 Introduction

A slot is 'single-valued' if it has a unique second argument, given the first. Relations (slots) can be declared as single-valued on the frame describing that relation using its cardinality slot (Section 4), giving it a value `N-to-1` or `1-to-1` as appropriate. If a slot is single-valued then KM can be sure that the multiply-inherited values for that slot are *coreferential*, ie. refer to the same individual. For example, consider the following KB:

```
(has-engine has
  (instance-of (Slot))
  (domain (Vehicle))
  (range (Engine))
  (cardinality (1-to-1)))       ; ie. a Vehicle has exactly one engine
                                ; (single-valued slot)

(every Vehicle has
  (has-engine ((a Engine with
                 (strength (*Powerful))))))

(Car has (superclasses (Vehicle)))
```

```
(every Car has
  (has-engine ((a Engine with
                 (size (*Average))))))
```

Slots can be declared as single-valued on the slot's frame, as shown above. If we now ask about the engine of a car, then we get two answers via multiple inheritance:

- a powerful engine (from `Vehicle`), and
- an average-sized engine (from `Car`)

As the slot `has-engine` has been declared as single-valued, KM concludes that these two descriptions are coreferential, and will thus combine them together. The syntactic process of combining the information is called *unification.* It involves merging the slot-values into a single frame, and redirecting future references to those (previously distinct) engines to point to that new frame:

```
KM> (showme (the has-engine of (a Car)))
(_Engine63 has                          ; Info has been unifed:
   (has-engine-of (_Car62))             ; _Engine63 is...
   (strength (*Powerful))               ; both powerful...
   (instance-of (Engine))
   (size (*Average)))                   ; *and* average size.
```

10.2.2 Lazy Unification

In general, full unification is expensive to compute: If the two frames being unified both have values for the same slot, then those values themselves must be recursively unified. In addition, a slot value may be a complex expression (eg. a path), not just an instance name. These expressions need to be evaluated in order to identify the instances being unified, which in turn may involve further multiple inheritance and unification. Finally, for complete unification, inherited as well as local information needs to be taken into account: In the above case, information from the `Engine` frame (also `Device`, `Physobj`, `Propulsion-device`, etc.) is also relevant to a car's engine, and should be included to obtain a complete description of the car's engine.

One approach, used in Description Logics, is to restrict the representation language's expressivity so (logically) complete concept descriptions can be computed. KM, however, uses a novel, alternative approach called *lazy unification*, in which slot-values are not immediately unified, but instead noted as 'to be unified' should the value of the slot be needed. For example, if the two instances being merged have expressions `expr1` and `expr2` for the same single-valued slot, then their unification is noted as `(expr1 & expr2)`, where `&` is the unification operator in KM. This delays actual unification of `expr1` and `expr2` until the value of that single-valued slot is actually needed (if ever). For example consider the following KB, where there are two values for the (single-valued) slot `fuel`:

```
(has-engine has
  (instance-of (Slot))
  (domain (Vehicle))
  (range (Engine))
  (cardinality (1-to-1)))

(fuel has
  (instance-of (Slot))
  (domain (Vehicle))
  (range (Gas-type))                ; eg. Philips'66 unleaded octane 87
```

```
  (cardinality (1-to-1)))

(every Vehicle has
  (has-engine ((a Engine with
                 (strength (*Powerful))
                 (fuel     ((a Gas-type with (combustibility (*Hi)))))))))

(Car has (superclasses (Vehicle)))
(every Car has
  (has-engine ((a Engine with
                 (size (*Average))
                 (fuel ((a Gas-type with (type (*Unleaded)))))))))
```

Now asking for the engine of a car causes these two engine descriptions to be lazily unified. The following shows the lazily unified result:

```
KM> (showme (the has-engine of (a Car)))
(_Engine66 has
   (has-engine-of (_Car65))
   (strength (*Powerful))
   (instance-of (Engine))
   (size (*Average))
   (fuel ((  (a Gas-type with (type (*Unleaded)))          ; note unification is
           & (a Gas-type with (combustibility (*Hi)))))))  ; deferred here
```

Note the value of the `fuel` slot is stored as the (still to be done) unification of the two expressions from the `Car` and `Vehicle` frames. They will only be unified when the value of fuel is actually requested, as we can now illustrate:

```
KM> (showme (the fuel of _Engine66))    ; now request the fuel slot's value
(_Gas-type68 has
   (fuel-of (_Engine66))
   (combustibility (*Hi))
   (instance-of (Gas-type))
   (type (*Unleaded)))

KM> (showme _Engine66)
(_Engine66 has
   (has-engine-of (_Car65))
   (strength (*Powerful))
   (instance-of (Engine))
   (size (*Average))
   (fuel (_Gas-type68)))           ;   <--- the now-unified gasoline type
```

This idea of lazy unification is fundamental to enabling information to be integrated together in a computationally tractable way in an expressive representation language.

10.2.3 The Unification Operator `&`

As the unification operator is part of the KM language, it can be issued directly from the KM query prompt as well, as illustrated below. KM also includes the operator `==` as a synonym for `&`.

;;; "Unify a green pet and a small fish."
;;; $\exists p, f\ isa(p, Pet) \wedge color(p, *Green) \wedge isa(f, Fish) \wedge size(f, *Small) \wedge p = f$

```
KM> ((a Pet with (color (*Green))) & (a Fish with (size (*Small))))  ; a Pet Fish
(_Pet70)

KM> (showme _Pet70)
(_Pet70 has
   (size (*Small))
   (instance-of (Pet Fish))
   (color (*Green)))

;;; “Fred is a person who is 40 years old.” (== is a synonym for &)
KM> (*Fred == (a Person with (age (40))))

KM> (the age of *Fred)
(40)
```

Named instances are assumed mutually exclusive (unique names assumption), and thus are non-unifiable. Anonymous instances, however, are not subject to this assumption and thus may be unified with other anonymous instances, or with named instances:

```
KM> (*Red & *Green)
ERROR! Failed to calc (*Red & *Green)!
NIL

;;; “Create a color.”
;;; ∃c isa(c, Color) ∧ ANS = c
KM> (a Color)
(_Color75)

;;; “That color is red.”
;;; _Color75 = *Red
KM> (_Color75 & *Red)
(*Red)

KM> _Color75
(*Red)                             ; _Color75 now points to *Red
```

Note that `=` and `&` are different in KM: `=` *tests* equality, while `&` (and its synonym `==`) *enforces* equality. `&` is a directive (“Unify these.”), rather than a question (“Do these unify?”). To test if unification (equality) is possible the query `&?` should be used instead:

```
;;; “Things can only have one color.”
;;; ∀x, c, c′ color(x, c) ∧ color(x, c′) → c = c′
KM> (color has (instance-of (Slot)) (cardinality (N-to-1)))

;;; “Can a red car and a green car be coreferential?”
KM> ((a Car with (color (*Red))) &? (a Car with (color (*Green))))
NIL                                                ; ie. No!
```

Again note that `&?` tests where equality would be logically consistent, while `=` tests whether equality holds.

10.2.4 Additional Value Unification Operators

KM uses some additional value unification operators, primarily for use internally to KM rather than by the user.

The first is `&+`, used internally within KM to heuristically decide whether two instances are coreferential or not when unifying sets (`&&`, see Section 10.3). Formally, `&+` is the same as `&` except with an additional requirement for successful unification, namely that either *instance1*'s immediate classes subsume *instance2*'s immediate classes, or vice versa. Thus

```
KM> ((a Cat) &+ (a Animal))
(_Cat1)
```

will succeed (given `Animal` is a superclass of `Cat`), but

```
KM> ((a Cat) &+ (a Pet))
NIL
```

will fail. If successful, the unification is returned, otherwise NIL.

`&+` is only used internally by `&&` (Section 10.3) to heuristically decide whether to treat members of two different sets, being unified together, as coreferential or not. It is not for user use, however the user may see it in the trace messages when debugging in detail (the +A or +U options).

The second is `&!`, meaning "eager unification". As the name suggests, unifying two instances with `&!` *will* recursively unify the instances' slot values immediately, rather than lazily delaying evaluation as described earlier.

Finally, for unifying all values in a set together, the special slot `unification` can be used:

```
KM> (the unification of (:set _Cat1 _Cat2 _Cat3))
(COMMENT: (_Cat1 & _Cat2) unified to be _Cat1)
(COMMENT: (_Cat1 & _Cat3) unified to be _Cat1)
(_Cat1)
```

10.3 Set Unification, Multi-valued Slots, and the Unification Operator `&&`

10.3.1 Unifying Sets

For multiply inherited values for multivalued slots, unification is more complex, as KM can no longer deductively conclude which values are coreferential. For example consider:

;;; "Every person owns a car."
;;; $\forall p \; isa(p, Person) \rightarrow \exists c \; isa(c, Car) \wedge owns(p, c)$
```
(every Person has
  (owns ((a Car))))

(Professor has (superclasses (Person)))
```

;;; "Every professor owns a car."
;;; $\forall p \; isa(p, Professor) \rightarrow \exists c \; isa(c, Car) \wedge owns(p, c)$
```
(every Professor has
  (owns ((a Car))))
```

Now do professors own one or two cars? One approach would be to assume all values are distinct; however, this defeats the central goal of integrating information from multiple generalizations together. Instead KM adopts a heuristic policy for unifying sets of instances together (each set coming from the same slot on a different frame):

- KM will try to unify instances from one set with instances from another set. A value from one set can unify with at most one value in the other set. Values *within* a set are not unified together.

- As unification is lazy, this check for valid unification is only partial. Unification will fail under the following conditions:
  - If the two instances being unified have different, named values for the same single-valued slot.
  - If the generalizations (classes) of one instance do not subsume or equal the generalizations of the other instance. This subsumption requirement is only used for unifying sets.
  - If the unification would violate some declared constraint (Section 12).

  Otherwise, unification will succeed and be carried out.

The set unification operator which implements this heuristic has the symbol `&&`. Its arguments must be a *set* of instances (or expressions which evaluate to instances), not single instances. The operator `===` is a synonym for `&&`.

;;; $\exists c, d, d', e \; isa(c, Cat) \wedge isa(d, Dog) \wedge isa(d', Dog) \wedge isa(e, Elephant) \wedge ANS = \{c \; d\} \cup \{d' \; e\}$[11]

```
KM> (((a Cat) (a Dog)) && ((a Dog) (a Elephant)))
(_Cat98 _Dog100 _Elephant101)          ; just Dog unifies

KM> (((a Cat)) && ((a Cat)))
(_Cat105)

;;; "Things can only have one color." (for illustration purposes)
KM> (color has
        (instance-of (Slot))
         (cardinality (N-to-1)))

KM> (((a Cat with (color (*Red)))) && ((a Cat with (color (*Blue)))))
(_Cat106 _Cat107)              ; won't unify as incompatible colors

KM> (the color of _Cat106)
(*Red)

KM> (the color of _Cat107)
(*Blue)
```

`&&` also plays a key role in handling multiple inheritance. Just as KM uses `&` to lazily unify values of single-valued slots with multiple inheritance, it uses `&&` to lazily unify values of multi-valued slots. For example:

```
(every Animal has
  (parts ((a Head)
          (a Body with (covering (*Skin))))))

(Mammal has (superclasses (Animal)))

(every Mammal has
  (parts ((a Leg) (a Leg) (a Leg) (a Leg))))

(Dog has (superclasses (Mammal)))
```

---

[11] Note KM's implementation of union includes identifying coreferences, as described.

```
(every Dog has
  (parts ((a Tail) (a Body with (covering (*Skin *Fur))))))
```

thus:

```
KM> (the parts of (a Dog))
(_Tail126 _Body133 _Leg128 _Leg129 _Leg130 _Leg131 _Head132)

KM> (showme _Body133)
(_Body127 has                                 ; merged info from Animal and Dog
   (parts-of (_Dog125))
   (instance-of (Body))
   (covering (((*Skin *Fur) && (*Skin)))))      ; note unification of this
                                              ; substructure is deferred
```

Note that the `Body` part of the animal and the `Body` part of the dog have been unified together when the multiple values were collected. Note also that unification of the two sets of values for the body's covering, namely `(*Fur)` and `(*Skin *Fur)`, has been delayed by the lazy unification mechanism, and is instead stored as `((*Skin *Fur) && (*Skin))`. A further query for the `covering` will cause this expression to be evaluated and those values unified:

```
KM> (the covering of (thelast Body))
(*Skin *Fur)
```

KM's heuristic 'guess' at which values are meant to be coreferential is the only non-deductive part of its inference. If it makes a bad guess, for example where delayed unification is later computed and fails, then KM has no means of recovering (there is no backtracking mechanism where the heuristic unification can be undone). However, the intension is that the knowledge engineer author the KB such that ambiguity does not arise (for example, unification can be prevented by tagging the not-to-be-unified items with a single-valued slot containing different identifiers). KM's heuristic set unification algorithm can thus be viewed as simply a short-hand to save the knowledge engineer the cumbersome job of explicitly noting coreference in the KB.

Finally, just as `&?` tests whether instance unification is possible, so `&&?` tests whether set unification is possible. This may seem a redundant test, as two sets can apparently always be unioned. However, as we describe later in Section 12.2, we can also specify constraints on sets in KM (eg. how many instances of a class it can contain), thus introducing conditions under which set unification may fail.

10.3.2 Controlling Set Unification

KM's set unification operator provides some heuristic "intelligence" in reasoning, but, as it is an unsound operator, can make mistakes on occasion. In particular, if values are iteratively added to a slot, and those values are of the same type as an existing value on that slot, KM will unify them:

```
KM> (_Person1 has (agent-of (_Move1)))
KM> (_Person2 has (agent-of (_Move2)))
KM> (_Person1 & _Person2)
KM> (showme _Person1)
(_Person1 has
  (agent-of (((_Move1) && (_Move2)))))   ; unification of the two Moves will be attempted
KM> (the agent-of of _Person1)
(COMMENT: (_Move1 && _Move2) unified to be _Move1)
(_Move1)
```

In the above case, KM has assumed coreferentiality of the two moves, and unified them.

One way of blocking this behavior is to define this slot as a `combine-values-by-appending` slot. This declares that new instances for a slot should be appended, rather than heuristically unified, with the existing values:

```
KM> (reset-kb)
KM> (agent-of has
      (instance-of (Slot))
      (combine-values-by-appending (t)))
KM> (_Person1 has (agent-of (_Move1)))
KM> (_Person2 has (agent-of (_Move2)))
KM> (_Person1 & _Person2)
KM> (showme _Person1)
(_Person1 has
  (agent-of (_Move1 _Move2)))          ; the two moves are not heuristically combined
KM> (the agent-of of _Person1)
(_Move1 _Move2)
```

Note that in this latter case, the two moves are *not* heuristically unified. This modified behavior only applies to combining atomic slot-values together – KM will still heuristically unify expressions (e.g., `(a Move)`) with values on this slot, to prevent infinite "growth" of values on a slot.

There are additional mechanisms also which can be used to controlling unification, in particular by tagging instances with identifiers to distinguish them, described later in Section 20, and making inequality assertions, described next.

10.3.3 Additional Set Unification Operators

Analogously to the additional value unification operators, the operator `&&!` unifies sets together eagerly (rather than lazily), and the special slot `set-unification` allows an arbitrary set of individual values to be heuristically unified using `&&`:

```
KM> (a Cat)
(_Cat5)
KM> (a Dog)
(_Dog6)
KM> (the set-unification of (:set (a Cat) _Dog6 _Cat5 (a Cat) (a Mouse)))
(COMMENT: (_Cat7 && _Cat5) unified to be _Cat7)
(COMMENT: (_Cat7 && _Cat8) unified to be _Cat7)
(_Cat7 _Dog6 _Mouse9)
```

10.4 Inequality

To assert that two instances are not equal, use the operator `/==`, e.g.:

```
;;; "_Color3 is definitely not green"
KM> (_Color3 /== *Green)

;;; "Is _Color3 unifiable with *Green?"
KM> (_Color3 &? *Green)
NIL
```

Internally, the inequality is recorded on the special slot `/==` on `_Color3`'s frame:

```
KM> (showme _Color3)
(_Color3 has
  (instance-of (Color))
  (/== (*Green)))
```

This slot is checked when unification is attempted.

Remember that two named (non-Skolem) instances are automatically assumed (and enforced) to be not equal, as KM makes the unique names assumption for named instances (Section 10.2.3). It would thus be redundant to assert, for example, `(*Red /== *Blue)`.

11 Partitions and Mutually Exclusive Classes

In many cases, a set of sibling classes are exhaustive and mutually exclusive (eg. the set of two classes {`Physical-Object`, `Nonphysical-Object`}). Such sets are called *partitions*. KM will not unify instances of different classes in a partition, as (by definition) an instance cannot belong to more than one of these classes at the same time.

A partition is declared in a KB as an instance of the class `Partition`, whose `members` are the classes in the partition. For example:

```
;;; "An instance cannot be both a cat and a dog."
;;; ∀c, d isa(c, Cat) ∧ isa(d, Dog) → c ≠ d
KM> (a Partition with (members (Cat Dog)))
(_Partition120)

;;; "Try unifying a cat and a dog..."
KM> ((a Cat) & (a Dog))
ERROR! Unification (_Cat121 & _Dog122) failed!
NIL
```

Note that without declaring the partition, KM would have instead undesirably allowed the two instances to unify (just as it desirably unified `((a Pet) & (a Fish))` earlier):

```
;;; Reset the KB (remove the above partition)
KM> (reset-kb)

;;; "Unify a cat and a dog."
KM> ((a Cat) & (a Dog))
(_Cat123)                              ; Unification undesirably succeeded,
                                       ; as the partition was not declared.
KM> (showme _Cat123)
(_Cat123 has
  (instance-of (Cat Dog)))             ; Answer: a Cat-Dog (!)
```

Named instances are automatically assumed mutually exclusive in KM, due to KM's unique names assumption, so there is no need to define a "partition of instances". It is not possible to specify directly in KM that anonymous instances are mutually exclusive (ie. non-unifiable), although this can be done indirectly by either placing them in mutually exclusive classes, or giving them different, named values for some single-valued slot.

12 Constraints

As well as placing values on slots, the user can declare *constraints* on the values which a slot can have. Some care is needed to understand how KM reasons with constraints: KM does not include a constraint reasoning engine for solving multiple constraints. Rather, in general, KM only *checks* constraints to make sure they hold, and reports an error if not (although a few constraints do have assertional import also, namely `must-be-a`, `excluded-values`, and `<>`, described shortly). There are two kinds of constraints:

Value Constraints: Apply to every *individual value* on the slot.
Set Constraints: Apply to the *set of values* on the slot.

Constraint expressions are placed on a frame's slot just like other expressions. However, they will be removed from any answer returned to the user.

Constraints play three roles in KM. First, they are a debugging tool to help the knowledge engineer. Second, KM will use knowledge of constraints to help determine whether two values can be unified or not, and constraints can be used to prevent unifications that would otherwise happen. Finally, for three special cases of a constraint, KM will do more than just check the constraint, but also make an assertion based on that constraint. These three special cases and their assertional import, explained in more detail shortly, are:

`(must-be-a` *class*`)`	will coerce all instances to be in *class*
`(exactly 1` *class*`)`	if multiple instances of *class*, they will be unified
`(at-most 1` *class*`)`	if multiple instances of *class*, they will be unified

Note that constraints are checked during inference, not at load-time. As a result, KM may not spot constraint violations at the moment they are introduced (as the below examples illustrate). Also, use of constraints will slow down unification in KM, as KM will check that constraints are not violated by unification. Finally, note that KM will not spot mutually inconsistent constraints during unification, only constraints which are violated by values or value sets.

12.1 Value Constraints

The following expressions declare constraints on individual slot-values. *Every* slot value must satisfy them:

Expression	**Meaning**
`(must-be-a` *class [*`with` *slotsvals]*`)`	Every value must be in *class* and have *slotsvals*. If this condition does not hold but could consistently hold, KM will enforce this condition by coercing each value to be in this class and have these properties.
`(mustnt-be-a` *class [*`with` *slotsvals]*`)`	Every value must *not* be in *class* and have *slotsvals*.
`(possible-values` $val_1 \ldots val_N$`)`	Each value must either be one of $val_1 \ldots val_N$, or be coercible to one of $val_1 \ldots val_N$ through specialization/unification. Note that `_Thing21` would be considered consistent with `(possible-values *Red *Blue)`, as `_Thing21` could be unified with either `*Red` or `*Blue` later in reasoning. As always val_i can also be an expression, which will be evaluated before the test. If the expression evaluates to multiple values, each is considered a possible value for the slot filler.
`(excluded-values` $val_1 \ldots val_N$`)`	Each value must not be one of $val_1 \ldots val_N$.
`(<>` *expr*`)`	The value must *not* equal (the evaluation of) *expr*. This is a special case of `excluded-values`.
`(constraint` *expr*`)`	General form. *expr* must be true for every value, where the keyword `TheValue` in *expr* denotes the value being tested.

For example:

;;; "A person's (possibly zero) friends are people."
;;; $\forall p\ isa(p, Person) \rightarrow (\ \forall p'\ friends(p, p') \rightarrow isa(p', Person)\ )$

```
KM> (every Person has
      (friends ((must-be-a Person))))
```

;;; "You cannot marry yourself."
;;; $\forall p\ isa(p, Person) \rightarrow (\ \forall p'\ spouse(p, p') \rightarrow p \neq p'\ )$

```
KM> (every Person has
      (spouse ((<> Self)))
```

;;; "Fred's spouse is himself." (a constraint violation)
;;; $spouse(*Fred, *Fred)$

```
KM> (*Fred has (spouse (*Fred)))
(*Fred)
```

;;; "Who is Fred's spouse?"

```
KM> (the spouse of *Fred)
ERROR! Constraint violation! Discarding value *Fred (conflicts with (<> *Fred))
NIL
```

Note that constraint violations are detected at inference time, not at load time, as this last example illustrates.
An example of the `possible-values` and `excluded-values` is:

;;; "Fred's possible favorite color(s) are red, blue, and green, and not pink." ;;; $\forall c\ favorite\text{-}colors(*Fred, c) \rightarrow c \in \{*Red, *Blue, *Green\} \wedge c \notin \{*Pink\}$

```
KM> (*Fred has
       (favorite-colors ((possible-values *Red *Blue *Green)
                         (excluded-values *Pink))))
```

Implementationally, these constraints are passed to all instances which appear on the slot `favorite-colors`, and stored on the two special slots `==` and `/==` (Section 10.4) on those instances.

In fact, these value constraint expressions are special cases of the more general value constraint expression `(constraint` *expr*`)`, meaning that *expr* must hold for every value in the slot. The keyword `TheValue` is used to refer to the value being tested, similar to the use of `It` in `forall` expressions. Thus, for example:

`(must-be-a Person)`	is shorthand for `(constraint (TheValue &? (a Person)))` (i.e., the value is unifiable with a person)
`(<> *Sue)`	is shorthand for `(constraint (TheValue /== *Sue))`

An example of the general form is shown below:

```
;;; "A person's children must all be at least 12 years younger than him/herself."
;;; ∀p isa(p, Person) → ( ∀c children(p, c) → isa(c, Person) ∧
;;;                                              (∀a, a′ age(c, a) ∧ age(p, a′) → (a < a′ − 12)))
KM> (every Person has
      (age ((a Number) (exactly 1 Number)))           ; 'exactly' explained shortly
      (children ((must-be-a Person with
                   (age ((constraint ((if   (    (numberp TheValue)
                                                and (numberp (the age of Self)))
                                         then (TheValue < ((the age of Self) - 12))
                                         else t)))))))))
```

Note that the constraint has been written to automatically succeed if the two ages being compared are not known quantities (e.g., `_Number23`).

```
KM> (*Sue has  (instance-of (Person)) (age (20)))
KM> (*Fred has (instance-of (Person)) (age (10)))

;;; "Fred is Sue's child."  (is a constraint violation)
KM> (*Sue has
       (children (*Fred)))

;;; "How old are Sue's children?"  (→ constraint violation encountered and reported)
KM> (the age of (the children of *Sue))
ERROR! Constraint violation! Discarding value *Fred (conflicts with (must-be-a ...
NIL
```

12.2 Set Constraints

The following expressions declare constraints on the *set* of values for a given slot:

Expression	**Meaning**
`(at-least` *n class*`)`	At least *n* instances in the set must be in *class*
`(at-most` *n class*`)`	At most *n* instances in the set must be in *class*
`(exactly` *n class*`)`	Exactly *n* instances in the set must be in *class*
`(set-constraint` *expr*`)`	General form. *expr* must be true for the set, where the keyword `TheValues` in *expr* denotes the set.

These constraints, like others, should be placed on slot values. KM only checks that the *upper* bound on the number of allowed values has not been exceeded, on the grounds that new values may be added later by the user or through inferencing. Thus if there are less values than required, then no error is reported. Hence, KM performs no test at all for `at-least` (it is a dummy constraint), as this is a lower bound constraint. For the special case where the upper bound is 1 (i.e., `(at-least/exactly 1` *class*`)`) KM will forcibly unify values together if there are more than one. If the upper bound is more than 1, and that bound has been exceeded, KM will report an error rather than try to correct the problem by unifying values together.

Be default, for the lower bound constraints, KM will not "pad" slots with new, additional instances if there are not enough present. For example, `(exactly 1` *class*`)` will test for but not cause creation of an instance of *class*. As a result, the user should not rely on constraints as a substitute for making regular assertions[12] . An example follows

;;; "Airplanes have a fuselage, exactly two wings, and at least one engine."
;;; $\forall a\ isa(a, Airplane) \rightarrow \exists f, w_1, w_2, e$
;;; $isa(f, Fuselage) \wedge isa(w_1, Wing) \wedge isa(w_2, Wing) \wedge isa(e, Engine)$
;;; $\wedge\ |\{n|parts(a,n) \wedge isa(n, Engine)\}| \geq 1\ \wedge\ |\{w|parts(a,w) \wedge isa(w, Wing)\}| = 2$[13]

```
KM> (every Airplane has
          (parts ((a Fuselage)
                  (a Wing with (side (*Left)))
                  (a Wing with (side (*Right)))
                  (a Engine)
                  (at-least 1 Engine)
                  (exactly 2 Wing))))
```

Note that we define both the presence of an engine `(a Engine)` and the constraint that there must be at least one `(at-least 1 Engine)`.

The general form `set-constraint` is used to define general constraints on the value set, where the keyword `TheValues` denotes the set. The following equivalences hold:

```
(at-most n class)   =  (set-constraint (the number of
                              (allof TheValues where (It isa class))) <= n))
(exactly n class)   =  (set-constraint (the number of
                              (allof TheValues where (It isa class))) <= n))
(at-least n class)  =  (ignored)
```

As already mentioned, note the built-in constraints check for provable violations rather than partial satisfactions of the constraints, and thus only the upper bounds on number constraints are checked. (This is why tests for `at-most` and `exactly` are the same, and the constraint `at-least` is ignored). The user can also declare his/her own set constraints using a `(set-constraint ...)` expression, for example the below illustrates a `set-constraint` which enforces "exactly one" in a stricter way, ensuring that the instance is not just allowed, but has actually been identified.

;;; "There must be exactly one left wing."
;;; $\forall a\ isa(a, Airplane) \rightarrow |\{\ w\ |\ parts(a,w) \wedge isa(w, Wing) \wedge side(w, *Left)\ \}| = 1$

---

[12] As a somewhat experimental mechanism, this default (lack of) behavior can be changed by setting the global variable `*max-padding-instances*` (default 0) to a positive integer, causing KM to creating "missing" instances on a slot up to a maximum (namely the smaller of the lower bound or `*max-padding-instances*`). The role of `*max-padding-instances*` is to avoid excessive instance generation with constraints like `(at-least 10000 Nucleotide)`.

[13] As mentioned, KM in fact performs a weaker test than this as the sets may only be partially known, only testing the upper bounds on the two number constraints.

```
KM> (every Airplane has
      (parts ((set-constraint
                 ((the number of (allof TheValues
                                    where (    (It isa Wing)
                                           and ((the side of It) = *Left)))) = 1)))))
```

12.3 Switching Off Constraint Checking

Some constraints may be primarily to help the knowledge engineer debug the knowledge base at KB authoring time, by implementing various consistency checks. However, once the author is confident the KB is correct, it may be desirable to switch those constraints off, so that run-time inferencing is faster (though at the risk that these "sanity checks" are not performed). To flag these constraints, a `(sanity-check ...)` wrapper can be used to rewrite expressions of the form:

```
(constraint expr)
(set-constraint expr)
```

as

```
(constraint (sanity-check expr))
(set-constraint (sanity-check expr))
```

For example:

```
(every Person has
   (age ((constraint (TheValue < 150)))))
```

would become

```
(every Person has
   (age ((constraint (sanity-check (TheValue < 150))))))
```

By default, sanity checking is "off", and so KM ignores these constraints. However, checking of these constraints can be turned on by:

```
KM> (sanity-checks)
```

and off again with

```
KM> (no-sanity-checks)
```

Note that the `(sanity-check ...)` wrapper cannot be placed around the special constraints such as `(must-be-a ...)`.

13 Sets, Sequences, and Bags

KM includes features for manipulating sets (unordered, no duplicates), sequences (ordered, duplicates allowed), and bags (unordered, duplicates allowed). Sequences are particularly important for text generation (Section 16), where the ordering of text fragments needs to be preserved. Bags are particularly important for arithmetic (Section 14), where duplicates in a group of numbers to operate on should *not* be removed. In general, KM tries to retain the ordering of elements of sets also, but the preservation of order is not guaranteed.

A set, sequence, or bag is expressed using a list starting with the keyword `:set`, `:seq`, or `:bag`. Sequences and bags are treated as *single* items in KM, while a set is treated as a *collection* of multiple items. Note that KM always flattens sets[14], and will drop the `:set` keyword if it is redundant (in particular, when returning the results of a KM query, which is already assumed to be a set).

In addition, the special symbol `:pair` is used to denote a sequence of length two.

13.1 Sets

Despite that KM does not guarantee that the ordering of values in a set will be preserved, the following functions allow the user to select a particular element by location in a set. Use of these functions is not recommended due this lack of guarantee.

`(the first of` *expr*`)`	get first element
`(the second of` *expr*`)`	get second element
`(the third of` *expr*`)`	get third element
`(the fourth of` *expr*`)`	get fourth element
`(the fifth of` *expr*`)`	get fifth element
`(theNth` *n* `of` *expr*`)`	get *n*th element
`(the last of` *expr*`)`	get last element

For example:

```
KM> (the first of (:set "a" "b" "c"))
("a")

KM> (theNth 3 of (:set "a" "b" "c"))
("c")
```

The function `(the number of` *expr*`)` returns the cardinality of a set:

```
KM> (the number of (:set "a" "b" "c"))
(3)
```

In addition, KM has the operators `includes` and `is-superset-of`, testing set membership and set-subset relationships respectively. See Section 7 earlier for examples of these.

13.2 Sequences and Bags

1. To access individual elements of a sequence or bag, use:

`(the1 of` *seq*`)`	get first element
`(the2 of` *seq*`)`	get second element
`(the3 of` *seq*`)`	get third element
`(theN` *n* `of` *seq*`)`	get *n*th element

   Note that these are not recommended for bags, as the ordering of elements in a bag is not guaranteed.

   ```
   KM> (the1 of (:seq 1 2 1 2 3))
   (1)

   KM> (theN 5 of (:seq 1 2 1 2 3))
   (3)
   ```

---

[14] This built-in behavior derives from the fact that each KM expression return a set of values, and hence to properly collect values from a set of expressions (e.g., as on a slot), the (set of sets) results must be flattened.

2. To convert a set into a bag or seq, use:

   ```
   (the seq of expr)
   (the bag of expr)
   ```

   For example:

   ```
   KM> (the bag of (the students of *Bruce))
   ((:bag *Fred *Joe))

   KM> (the bag of (:set 1 2 3))
   ((:bag 1 2 3))
   ```

3. To convert a bag or seq back into a set, use:

   ```
   (the elements of bagseq)
   ```

   For example:

   ```
   KM> (the elements of (:bag 1 2 3))
   (1 2 3)

   KM> (the elements of (:bag 1 1 2 3))
   (1 2 3)
   ```

4. To operate on all elements of a bag or seq, use:

   ```
   (the slot of bag)
   (the slot of seq)
   ```

   When the slot of a bag/seq is requested, *slot* is queried for all elements of the bag/seq, and a new bag/seq is returned. i.e., `(the` *slot* `of` *bagseq*`)` is mapped over all elements of the bag/seq.

   ```
   KM> (the age of (:bag *Chris *Kevin))          ; (twins)
   ((:bag 13 13))

   KM> (the age of (the bag of (the children of *Pete)))
   ((:bag 13 13 8))                               ; Chris, Kevin, Rachel
   ```

5. For conditional removal of elements, use:

   ```
   (forall-bag [var in] bag [where test] expr)
   (forall-seq [var in] seq [where test] expr)
   ```

   These functions are the bag/seq equivalent of the `forall` forms (Section 8), where *bag*, *seq* is an expression evaluating to a *single* bag/seq. These functions evaluate *expr* for each element of the bag/seq passing *test*, returning a new bag/seq containing the results of that evaluation. As with `forall`, either a variable *var* or the keyword `It` can be used to refer to each item being tested.

```
KM> (forall-bag (:bag 1 2 1 2)
          where (It < 2)
                It)
((:bag 1 1))

;;; Equivalent to the above
KM> (forall-bag ?x in (:bag 1 2 1 2)
                where (?x < 2)
                      ?x)
((:bag 1 1))

KM> (forall-seq (:seq 1 2 3 3) (It + 1))
((:seq 2 3 4 4))

KM> (forall-seq (:seq 1 2 3 3) where (It >= 2) (It + 1))
((:seq 3 4 4))
```

6. For "lengths" (i.e. number of elements) of seq/bags, use:

   ```
   (the seq-length of seq)
   (the bag-length of bag)
   ```

   Note that `(the length of` *seq*`)` is *not* the correct function: A bag/seq is considered a single (structured) item in KM, and so `(the length of` *seq*`)` will return the answer 1. In contrast, `seq-length` and `bag-length` "open up" the bag/seq to count the elements within it.

   ```
   KM> (the bag-length of (:bag 1 2 1))
   (3)
   ```

7. To combine two seqs or bags, use the infix operator:

   ```
   (seq append seq)
   (bag append bag)
   ```

   or for multiple seqs/bags, use `append` as a slot to append a sequence of sequences (or a bag of bags):

   ```
   (the append of (:seq seq1 seq2 ... seqn))
   (the append of (:bag bag1 bag2 ... bagn))
   ```

   This is analogous to the infix/prefix operators used for arithmetic (e.g., `+` and `(the sum of ...)`).

   ```
   KM> ((:seq 1 2) append (:seq 2 3))
   ((:seq 1 2 2 3))

   KM> (the append of (:seq (:seq 1 2) (:seq 2 3)))
   ((:seq 1 2 2 3))
   ```

8. The elements in a sequence can be reversed using:

   ```
   (reverse seq)
   ```

   For example:

```
KM> (reverse (:seq 1 2 3))
((:seq 3 2 1))
```

13.3 Equality of Sets, Sequences, and Bags

KM respects the definitions of sets, sequences, and bags when testing equality, as illustrated below:

```
KM> ((:set 1 2 1) = (:set 2 1))
(t)

KM> ((:bag 1 2 1) = (:bag 2 1 1))
(t)

KM> ((:bag 1 2 1 2) = (:bag 2 1 1))
NIL
```

13.4 Unification of Sets, Sequences, and Bags

Sets are unified using the set unification operator `&&`, as described earlier in Section 10.

Unlike sets, a sequence or bag is treated as a single entity, rather than multiple entities. As a result, sequences and bags are unified with the value unification operator `&`. When sequences are unified, KM matches and unifies elements pairwise:

```
KM> ((:seq 1 2 3) & (:seq _X 2))
(COMMENT: (1 & _X) unified to be 1)
((:seq 1 2 3))
```

Bags are currently unified as if they were sequences, a limitation of the current implementation. Thus KM will not realize that two bags containing the same elements in a different order can be unified. Note that this limitation only affects unification (`==`, `&`), not equality testing (`=`).

14 Arithmetic

14.1 Arithmetic Operators

KM has five infix arithmetic operators for number manipulation: `+`, `-`, `/`, `*`, and `^` (exponentiation). As always, these expressions can be issued at the KM prompt, or used in the value of a frame's slot, for example:

```
KM> (1 + 2)
(3)

KM> (1 + (2 ^ 4))
(17)
```

`^` has highest precedence, followed by `*` and `/`, followed by `+` and `-`.

KM also has several built-in *aggregation slots*, for combining multiple values into a single value. These aggregation slots are applied to a bag or set of values, and return a single, combined value. They are useful when the number of values to be combined is not known in advance. (KM also allows the user to define additional aggregation slots, see Section 29.15). The built-in arithmetic aggregation slots are:

`(the sum of` *bagexpr*`)`	addition
`(the difference of` *bagexpr*`)`	subtraction (10 - 5 - 1 = 4)
`(the product of` *bagexpr*`)`	multiplication
`(the quotient of` *bagexpr*`)`	division (8 / 2 / 2 = 2)
`(the average of` *bagexpr*`)`	average
`(the max of` *bagexpr*`)`	maximum
`(the min of` *bagexpr*`)`	minimum
`(the number of` *expr*`)`	cardinality

Apart from `number`, these should all be applied to a KM bag of numbers (`:bag`), although `max` and `min` can also be applied to a set (KM will temporarily coerce the set to a bag). `number` should be applied to a set, and it returns the number of elements in a set. (If it is applied to a bag or sequence, it will return the answer `1`, i.e., there is 1 bag/sequence. To count the number of elements in a bag/sequence use `bag-length` and `seq-length` instead, Section 13.2).

In addition, the following slots are *mapping functions*, i.e., they are applied to a set/sequence/bag of numbers, and return a similar sized set/sequence/bag of new numbers:

`(the abs of` *expr*`)`	remove negative sign (eg. -1 becomes 1)
`(the floor of` *expr*`)`	remove decimals (eg. 1.02 becomes 1)
`(the log of` *expr*`)`	log (base e)
`(the exp of` *expr*`)`	exponent
`(the sqrt of` *expr*`)`	square root

14.2 Examples

14.2.1 Basic Usage

```
;;; "Joe's exam scores were 10, 32, and 31."
;;; exam-score(*Joe, 10) ∧ exam-score(*Joe, 32) ∧ exam-score(*Joe, 31)
KM> (*Joe has
       (instance-of (Person))
       (exam-scores ((:bag 10 32 31))))

;;; "What was Joe's best score?"
;;; max({ s | exam-scores(*Joe, s) })
KM> (the max of (the exam-scores of *Joe))
(32)

;;; "What was Joe's worst score?"
;;; min({ s | exam-scores(*Joe, s) })
KM> (the min of (the exam-scores of *Joe))
(10)

KM> (the sum of (the exam-scores of *Joe))
(73)

;;; sum({1 2})
KM> (the sum of (:bag 1 2))
(3)
```

14.2.2 Finding Item(s) with a Top Score

Here is an example of arithmetic being used to find the hottest city in the USA.

;;; "USA is a country, with Seattle and Austin as cities."
;;; $isa(*USA, Country) \wedge cities(*USA, *Seattle) \wedge cities(*USA, *Austin)$

```
KM> (*USA has
        (instance-of (Country))
        (cities (*Seattle *Austin)))            ; just use two cities for now!

KM> (*Seattle has
        (instance-of (City))
        (mean-temp (50)))

KM> (*Austin has
        (instance-of (City))
        (mean-temp (90)))
```

;;; "How hot is the hottest city in the USA?"
;;; $max(\{\ t \mid \exists c\ cities(*USA, c) \wedge mean\text{-}temp(c, t)\ \})$

```
KM> (the max of (the mean-temp of (the cities of *USA))) ; hottest temp
(90)
```

;;; "Which is(are) the hottest city(s)?"
;;; $\{\ c \mid cities(*USA, c) \wedge mean\text{-}temp(c, t) \wedge t = max(\{t' \mid \exists c'\ cities(*USA, c') \wedge mean\text{-}temp(c', t')\})\ \}$[15]

```
KM> (allof (the cities of *USA)                         ; the hottest city(s)
     where ((the mean-temp of It) =
        (the max of (the mean-temp of (the cities of *USA)))))
(*Austin)
```

14.2.3 Cardinality

Consider the KB:

;;; "Cars are vehicles"
;;; $superclasses(Car, Vehicle)$

```
(Car has (superclasses (Vehicle)))
```

;;; "Cars have four wheels, an engine, and a chassis."
;;; $\exists w_1, w_2, w_3, w_4, e, c\ isa(w_1, Wheel) \wedge isa(w_2, Wheel) \wedge isa(w_3, Wheel) \wedge isa(w_4, Wheel)$
;;; $\wedge\ isa(e, Engine) \wedge isa(c, Chassis)$

```
(every Car has
  (parts ((a Wheel) (a Wheel) (a Wheel) (a Wheel) (a Engine) (a Chassis))))
```

we can ask:

;;; "How many wheels does a car have?"

```
KM> (the number of (the Wheel parts of (a Car)))
(4)
```

Even better, this expression can be placed in the frame itself:

```
(Car has (superclasses (Vehicle)))
```

---

[15] Again, this semantics assumes mean temperature is single-valued.

```
;;; (Last line) "A car's wheel-count is the number of wheels on that car."
;;; ∀c isa(c, Car) → wheel-count(c, number({ w | parts(c, w) ∧ isa(w, Wheel) }))
(every Car has
  (parts ((a Wheel) (a Wheel) (a Wheel) (a Wheel) (a Engine) (a Chassis)))
  (wheel-count ((the number of (the Wheel parts of Self)))))
```

14.2.4 Proper use of bags and arithmetic

As the basic arithmetic operators act on bags, not sets, care is needed to convert sets into bags appropriately. Remember, KM removes duplicates from sets. Consider the example of a teacher and his class

```
KM> (*Alan has (class (*Class1)))                          ; the teacher
KM> (*Sue has (student-of (*Class1)) (age (18)))
KM> (*Joe has (student-of (*Class1)) (age (18)))
KM> (*Mike has (student-of (*Class1)) (age (21)))
...etc...
```

Suppose now we want to find the average age of students in Alan's class. Remember, we need a *bag* of the ages, so that duplicates are preserved. However, note the following two alternatives:

```
;;; Wrong! (the age of ...) returns a set (duplicates removed), which is then
;;; turned into a bag (i.e., too late!).
KM> (the bag of (the age of (the students of (the class of *Alan))))
((:bag 18 19 21 22))

;;; This is correct: Get a bag of the students, then find their ages.
KM> (the age of (the bag of (the students of (the class of *Alan))))
((:bag 18 18 21 18 21 18 18 21 ...))

;;; Correct way of finding the average age
KM> (the average of (the age of
                (the bag of (the students of (the class of *Alan))))
(19.4241)
```

14.3 Comparing Numbers: Precision and Tolerance

To tolerate the tiny errors which can arise when using floating point arithmetic, KM uses a parameter called `*tolerance*` (default value 0.0001) to determine when two numbers are "sufficiently close" to consider them equal when testing equality. The semantics of `*tolerance*` are as follows:

- For large numbers, it is treated as an absolute value of tolerance, i.e., if two large numbers are within +/- 0.0001 of each other, they are considered equal.
- For small numbers, it is treated as a relative value of tolerance, i.e., if two numbers are within +/- 0.01% of each other, they are considered equal.

The choice of which method to use depends on the numbers themselves, with the overall algorithm behaving as follows:

`x = y` **IF** `x` equals `y` +/- (0.0001 *or* 0.01% of max(`x`,`y`), whichever is smaller)

This algorithm is designed to support the following desired behavior:

$$0.00001 \neq 0.00002$$
$$4.99999 = 5.00000$$
$$499999 \neq 500000$$

The value of `*tolerance*` can be changed by a Lisp setq statement, if desired.

In addition, the user can explicitly state the tolerance to apply for specific equality tests using the following forms:

(*expr* = *expr* +/- *expr*) (absolute tolerance)
(*expr* = *expr* +/- *expr* %) (relative tolerance)

For example:

```
KM> (99 = 100 +/- 2)
(t)

KM> (99 = 100 +/- 1 %)
(t)
```

15 Reference by Description

15.1 The Object Stack

During reasoning, KM maintains a list (stack) of the instances created or encountered. We refer to this stack of instances as the object stack. The command `(show-obj-stack)` lists the stack, and `(clear-obj-stack)` clears it (but, note, *doesn't* remove the instances themselves from the KB):

```
KM> (show-obj-stack)
   _Chassis24
   _Engine23
   _Wheel22
   _Wheel21
   _Wheel20
   _Wheel19
   _Car18
   _Chassis17
   ...
```

The role of the stack is to provide a notion of "context" or "current scenario" for reasoning; while there may be many instances in the KB, reasoning can be restricted to just those instances in the stack. To refer to these instances in the current context by description, the commands `(the ...)` and `(every ...)` are used, as described below. This ability to reference objects by description ("content-addressable memory") is particularly useful for natural language processing applications, as reference resolution based on description is an important part of language understanding.

In addition, the expression `(thelast` *class*`)` refers to the most recent instance of *class* created. It is a useful convenience, in a top-level query, for referring back to an earlier instance without explicitly naming it (as illustrated earlier in this manual). This command should *never* be included in a KB, though, as its result is highly situation-specific.

15.2 Finding Instances

The forms (`the` *class [*`with` *slotsvals]*) and (`every` *class [*`with` *slotsvals]*) are used to find instances by description. These commands do *not* search the entire KB, but instead restrict their search to

1. Instances on the object stack
2. Instances "accessible from" other instances embedded in the query. For example, a query for `(every Person with (loves (*Sue)))` mentions the instance `*Sue` in the query, and hence KM has a "handle" to start searching from (i.e. start at `*Sue`, and find who she is the `loves-of`). As with all reasoning in KM, any instances encountered during the search are added to the object stack (whether or not the search is successful). If there are no explicitly named instances in the query, however, eg. the query `(every Person)`, KM will simply search the object stack.

Note also that if the user never issues a `(clear-obj-stack)` command, then the stack will contain all reified instances in the KB, and thus the search will, in this case, effectively search the whole KB.

15.2.1 Finding an Instance

To illustrate the form (`the` *class [*`with` *slotsvals]*), consider the small KB:

```
;;; "Joe is a person who owns three vehicles (two cars and a van)."
KM> (*Joe has
        (instance-of (Person))
        (vehicle ((a Car with (color (*Brown)) (age (*Old)))
                  (a Car with (color (*Red))   (age (*New)))
                  (a Van with (color (*Red))   (age (*Old))))))

KM> (Car has (superclasses (Passenger-Vehicle)))
KM> (Van has (superclasses (Passenger-Vehicle)))
KM> (*Red has (instance-of (Bright-Color)))
KM> (*Brown has (instance-of (Dark-Color)))
```

How can we refer to "Joe's brown car", or "Joe's old, red car"? Note that the path

`(the *Brown color of (the Car vehicle of *Joe))`

refers to the brown color of a car, not to the car itself. Instead, a `(the ... with...)` can be used. Just as the expression "(`a` *class* `with` *slotsvals*)" *creates* an instance of *class* with the given description, so "(`the` *class* `with` *slotsvals*)" *finds* the instance of *class* which matches (is subsumed by) the description, subject to the search constraints described above in Section 15.2. For example:

```
;;; "Which is the brown car owned by Joe?"
;;; ι(c)( isa(c, Car) ∧ vehicle-of(c, *Joe) ∧ color(c, *Brown) )[16]
KM> (the Car with
       (vehicle-of (*Joe))
       (color (*Brown)))
(_Car0)
```

---

[16] Here using iota notation, where $\iota(c)\alpha(c)$ denotes the unique instance for which formula $\alpha(c)$ (containing free variable c) is true. $\{\ \iota(c)\alpha(c)\ \} = \{\ c \mid \alpha(c)\ \}$ when $\alpha(c)$ is true of just a single instance, and is an ill-formed sentence otherwise. See [12, p47-48].

```
;;; Check: “What color is it?” (should be brown)
KM> (the color of _Car0)
(*Brown)                                    ; Yes! (it's the Brown one)

;;; “Find the person owning (at least one) red van.”
;;; ι(p)( ∃v isa(p, Person) ∧ vehicle(p, v) ∧ isa(v, Van) ∧ color(v, ∗Red) )
KM> (the Person with
         (vehicle ((a Van with
                      (color (*Red))))))
(*Joe)
```

Note that `the` assumes there will be exactly one answer, and will generate an error if none are found. Thus, this form should be used if the knowledge engineer expects a unique answer, as a consistency check.

15.2.2 Finding a Set of Instances

To collect multiple (possibly zero) answers, the form `(every ... with ...)` should be used instead. In fact, `(the ... with ...)` is implemented as `(every ... with ...)` plus an extra check that exactly one instance was found. The below illustrates the `every` form:

```
;;; “Find all the people owning (at least one) red van.”
;;; { p | ∃v isa(v, Van) ∧ color(v, ∗Red) ∧ vehicle(p, v) }[17]
KM> (every Person with
         (vehicle ((a Van with
                      (color (*Red))))))
(*Joe)

;;; “Find all the passenger-vehicles owned by Joe.”
;;; { v | isa(v, Passenger-Vehicle) ∧ vehicle-of(v, ∗Joe) ∧ color(v, ∗Red) }
KM> (every Passenger-Vehicle with
          (vehicle-of (*Joe))
          (color (*Red)))
(_Car1 _Van2)
```

Thus, just as `(every ... has ...)` is an *intensional* description of a classs, so `(every ... with ...)` computes the partial *extension* of a class (partial because only instances created thus far are returned).

15.3 Find-or-Create

The KM command `the+` behaves like `the`, except rather than generate an error if a matching instance cannot be found, it will create one. `the+` thus combines the functionality of `(the ...)` and `(a ...)`. This function is called *find-or-create*, also used in other KR languages such as Algernon [13]. In fact, `a` and `the+` have similar semantics (assert the existence of an instance and return it), except that `a` does not assume any coreference between that instance and others in the KB, while `the+` will, if such coreference is possible. If such coreference is ambiguous, e.g. two instances match a `the+` description, then KM will signal an error. The use of `the+` is illustrated below:

---

[17] KM may only return a subset of this set, due to the search constraints described in Section 15.2

```
;;; "Find-or-create Joe's red car."
;;; ∃c isa(c, Car) ∧ vehicle-of(c, *Joe) ∧ color(c, *Red) ∧ ANS = c[18]
KM> (the+ Car with
        (vehicle-of (*Joe))
        (color (*Red))))
(_Car1)                                  ; found the old one (defined earlier)

;;; "Find-or-create Joe's blue car."
;;; ∃c isa(c, Car) ∧ vehicle-of(c, *Joe) ∧ color(c, *Blue) ∧ ANS = c
KM> (the+ Car with
        (vehicle-of (*Joe))
        (color (*Blue))))
(_Car3)                                  ; a new one is created, [1]

KM> (showme *Joe)
(*Joe has
  (instance-of (Person))
  (vehicle (_Car3 _Van2 _Car1 _Car0)))    ; note new _Car3, from [1] above
```

Finally, the symbol `a+` is a synonym for `the+` in KM.

In the terminology of computational linguistics, the "find" and "create" operations correspond to "binding" and "accommodation" operations in reference resolution.

16 Text Generation

16.1 Introduction

Slot-values can include text strings, and this provides the basis for automatic text generation from the KB, using simple "fill in the blank" text templates. Sophisticated text generation capabilities were achieved in a previous project (project Halo, http://www.projecthalo.com/) using this mechanism.

A template is a sequence of strings and KM expressions, and is placed as the value of a frame's slot. (Note that a sequence rather than set is needed, to preserve order and allow repeated elements). Filling in the template involves evaluating the paths in the template, replacing the found instances with their print names, and then concatenating the resulting string fragments together. The first of these tasks, evaluating the paths, happens automatically if the user (or another part of the KB) queries for that slot's value (KM always evaluates slot values before returning them). The other two tasks, namely replacing the found instances with their print names and concatenating the results, is performed by the function `(make-sentence ...)` or `(make-phrase ...)`

16.2 Making Phrases and Sentences

To convert a sequence of fragments into a sentence, the expression `(make-sentence` *expr*`)` is used. `make-sentence` first evaluates all paths in *expr*, returning a sequence of elements (typically strings and instances). It then converts any instances to their text names (see next subsection) and concatenates all the elements, capitalizing the first word and adding a period. `make-phrase` does the same thing, except does not capitalize or add a period. For example:

---

[18] where, as usual, ANS is the returned answer (a function of the query + KB), see Section 3.2.

```
KM>  (make-sentence (:seq *Pete "is happy"))
("Pete is happy.")

KM>  (make-phrase (:seq *Pete "is happy"))
("pete is happy")
```

If the text template (sequence) itself contains a sequence, or an expression which evaluates to a sequence, then KM operates recursively, building a text string from each embedded sequence and then combining those together. For example:

```
KM>  (make-sentence (:seq "I think" (:seq "the" "man") "is happy"))
("I think the man is happy.")
```

This mechanism allows the user to embed expressions in a template which themselves return text templates.

By default, KM places a space character between each text string element. To suppress this, use the special string `"nospace"`:

```
KM> (make-phrase (:seq "a" "b"))
("a b")

KM> (make-phrase (:seq "a" "nospace" "b"))
("ab")

KM> (make-phrase (:seq "I like" Cat "nospace" "s"))
("I like cats")
```

16.3 Generating Names of Instances

As described above, a text generation sequence will evaluate to a sequence of strings and instances. To convert an instance to text, the built-in slot `name` is used, which returns either a string or a sequence of strings. By default, `(the name of` *instance*`)` behaves as follows:

- Named instances called `*`*name* are converted to the lower-case string `"`*name*`"`
- Anonymous instances (i.e., with names starting with '_') are converted to the sequence `(:seq "the"` *class*`)`, where *class* is the first of that instance's immediate generalizations.
- An instance of a class (e.g., `Cat`) is converted to a lower-case string (e.g., `"cat"`)

```
KM> (the name of *MyCar)
("mycar")

KM> (the name of (a Car))
((:seq "the" Car))

KM> (the name of Car)
("car")
```

This default behavior can be changed by declaring an explicit value on the `name` slot of an instance or one of its generalizations.

16.4 Converting Sets to Text

The function `(andify` *expr*`)` converts a set to a text sequence, inserting the word `"and"` and commas as appropriate:

```
KM> (andify (:set 1))
((:seq 1))

KM> (andify (:set 1 2))
((:seq 1 " and " 2))

KM> (andify (:set 1 2 3))
((:seq 1 ", " 2 ", and " 3))
```

As before, the function `make-sentence` can then be used to convert the sequence into a text string.

16.5 Example

Thus for complete text generation, we define a "text template" on a frame's slot, query it to find its components, and recursively apply `make-sentence` to the result to produce a single string. For example, consider a text template for describing "removing" activities:

```
(Remove has (superclasses (Activity)))

(every Remove has
  (text ((:seq "Remove" (the object of Self) "from" (the location of Self)))))
```

Thus a query for this slot on an instance of `Remove` will produce the instantiated template:

```
KM> (the text of (a Remove with
                        (object ((a Sample)))
                        (location ((a Box)))))
((:seq "Remove" _sample346 "from" _box347))
```

And a call to `make-sentence` will convert this to a single text string:

```
KM> (make-sentence (the text of (a Remove with
                                (object ((a Sample)))
                                (location ((a Box))))))
("Remove the sample from the box.")
```

The frame `Remove` thus includes information about how to describe itself. (We could add other information too, for example the preconditions and effects of a remove). Now we can use the `Remove` concept elsewhere, and have KM resolve the references to its `object` and `location` at run-time, and hence generate customized text.

16.6 Situation-Specific Text Generation

Consider a small KB about electrophoresis (a process for separating a chemical into its constituents):

```
(Remove has (superclasses (Activity)))

(every Remove has
  (text ((:seq "Remove" (the object of Self) "from" (the location of Self)))))

(every Electrophoresis has
  (sample ((a Chemical)))
```

```
  (equipment ((a Separation-unit) (a Syringe)))
  (first-task (
    (a Remove with
        (object ((the sample of Self)))
        (location ((the delivery-medium of (the sample of Self))))))))

(Albumin has (superclasses (Chemical)))

(every Albumin has
  (delivery-medium ((a Bottle))))
```

Note that the first-task of electrophoresis is described as a particular *instance* of a remove, in which the object is the sample being separated, and the location it's being removed from is that sample's delivery medium. We can now ask for a description of this remove:

```
KM> (the text of (the first-task of (a Electrophoresis with
                                       (sample ((a Albumin))))))
((:seq "Remove" _Albumin353 "from" _Bottle354))

KM> (make-sentence (the text of (the first-task of (a Electrophoresis
                                                   with (sample ((a Albumin)))))))
("Remove the albumin from the bottle.")
```

16.7 A More Detailed Example

A more sophisticated KB illustrating this is shown in Tables 2 and 3. Note that one of the tasks is conditional (on the density of the sample being ≥ 3), and that the duration of the `Wait` step is a function of the sample's density.

Here is an example of querying this KB. Note how the objects in the text are customized based on which sample is being separated (and, if this was a more general KB, other experimental details):

```
KM> (forall (the subevents of (a Electrophoresis with (sample ((a Albumin)))))
            (make-sentence (the text of It)))
("Remove the albumin from the bottle."
 "Insert the albumin into the separation-unit using the syringe."
 "Wait 36 seconds."
 "Remove the albumin from the separation-unit."
 "Store the albumin in the fridge.")

KM> (forall (the subevents of (a Electrophoresis with (sample ((a Endoprotein)))))
            (make-sentence (the text of It)))
("Remove the endoprotein from the box."
 "Insert the endoprotein into the separation-unit using the syringe."
 "Add the dilutant to the separation-unit."
 "Wait 125 seconds."
 "Remove the endoprotein from the separation-unit."
 "Store the endoprotein in the vacuum-flask.")
```

16.8 Printing Floating Point Numbers

End-users are often unhappy to see floating point numbers printed out to multiple decimal places. To provide some control of how numbers are printed, KM has a global variable called

```
;;; ======================= START OF DEMO KB ==============================

(every Electrophoresis has
  (sample ((a Chemical)))
  (equipment ((a Separation-unit) (a Syringe)))
  (subevents (
    (a Remove with
        (object ((the sample of Self)))
        (location ((the delivery-medium of (the sample of Self)))))
    (a Insert with
        (object ((the sample of Self)))
        (destination ((the Separation-unit equipment of Self)))
        (equipment ((the Syringe equipment of Self))))
    (if   ((the density of (the sample of Self)) >= 3)
     then ((a Add with
                (object ((a Dilutant)))
                (destination ((the Separation-unit equipment of Self))))))
    (a Wait with
        (duration ((the floor of ((the density of (the sample of Self)) * 30))))
        (units (*Seconds)))
    (a Remove with
        (object ((the sample of Self)))
        (location ((the Separation-unit equipment of Self))))
    (a Store with
        (object ((the sample of Self)))
        (destination ((the storage-medium of (the sample of Self))))))))

;;; ----------

(Remove has (superclasses (Activity)))
(every Remove has
  (text ((:seq "Remove" (the object of Self) "from" (the location of Self)))))

(Insert has (superclasses (Activity)))
(every Insert has
  (text ((:seq "Insert" (the object of Self) "into" (the destination of Self)
                "using" (Self equipment)))))

(Add has (superclasses (Activity)))
(every Add has
  (text ((:seq "Add" (the object of Self) "to" (the destination of Self)))))

(Wait has (superclasses (Activity)))
(every Wait has
  (text ((:seq "Wait" (the duration of Self) (the units of Self)))))

(Store has (superclasses (Activity)))
(every Store has
  (text ((:seq "Store" (the object of Self) "in" (the destination of Self)))))

;;;           ---------- CONTINUED OVER... ----------
```

Table 2: A simple KB, containing text templates for text generation (part 1).

```
;;;           ---------- DEMO KB (CONT) ----------

(Albumin has (superclasses (Chemical)))
(every Albumin has
  (density (1.2))
  (delivery-medium ((a Bottle)))
  (storage-medium ((a Fridge))))

(Endoprotein has (superclasses (Chemical)))
(every Endoprotein has
  (density (4.2))
  (delivery-medium ((a Box)))
  (storage-medium ((a Vacuum-flask))))

;;; ====================== END OF DEMO KB==============================
```

Table 3: A simple KB, containing text templates for text generation (part 2).

`*output-precision*`, determining how many decimal places `make-sentence` should print out. Note this does not affect KM's internal storage and reasoning with numbers, only the way numbers are displayed with `make-sentence` and `make-phrase`.

By default, `*output-precision*` has the value 2, meaning that `make-sentence` will print out numbers to two decimal places either in normal notation (if the number > 0) or scientific notation (if the number is < 0). The complete behavior is illustrated below:

Number	**Printed (by `make-phrase` or `make-sentence`) as:**[19]	
	`*output-precision*`=2	`*output-precision*`=3
123456.789	”123456.79”	”123456.790”
12345.6789	”12345.68”	”12345.679”
1234.56789	”1234.57”	”1234.568”
123.456789	”123.46”	”123.457”
12.3456789	”12.35”	”12.346”
1.23456789	”1.23”	”1.235”
0.123456789	”0.123”	”0.1235”
0.0123456789	”0.0123”	”0.01235”
0.00123456789	”1.23e-3”	”0.001235”
0.000123456789	”1.23e-4”	”1.235e-4”
0.0000123456789	”1.23e-5”	”1.235e-5”

To turn off this rounding function, do:

```
  KM> (SETQ *OUTPUT-PRECISION* NIL)
```

17 Automatic Classification

17.1 Introduction

KM includes machinery for handling *defined concepts*, ie. concepts with a set of sufficient conditions for class membership/instance equivalence. For class membership, if an instance satisfies the class's defining properties then KM asserts that it is an instance of that class. For instance equivalence, if an instance satisfies another instance's defining properties then KM asserts that both instances are coreferential (by unifying them, described in more detail in Section 10). KM uses these definitions to automatically classify instances during run-time query processing. However, note that KM does not check that definitions of different classes are consistent with each other, nor check that defined classes are correctly placed in the isa hierarchy.

Classification can be thought of as 'recognition' that a particular instance is a member of/coreferential with another concept in the KB, and hence allow additional information about that instance to be inferred. Like multiple inheritance, the ability to classify is central to building modular knowledge-bases: As separate pieces of information come together, the compound result may be recognizable as an instance of an already-known concept, and allow further information to be concluded. This in turn may allow further classification to occur etc. This cycle of classification and elaboration can be powerful, and is described in more detail in [14].

Definitional properties of a concept are placed on a *separate frame* for that concept, specified just like a normal frame except using a '`has-definition`' rather than '`has`' keyword. For indexing purposes, the definitional properties *must* include an `instance-of` slot, stating the most general class(es) which instances can have. A concept cannot have more than one definition.

```
(every <class> has-definition
   (instance-of (<most-general-class>))
   (<slot1> (<expr11> <expr12> ...))       ; conditions for membership
   ... )

(<instance> has-definition
   (instance-of (<most-general-class>))
   (<slot1> (<expr11> <expr12> ...))       ; conditions for equivalence
   ... )
```

A class's definitional properties inherit down to its instances, just as normal assertional `(every...has...)` properties do. Note it is redundant to state a property both as a definition and as an assertion, as the definition necessarily implies the assertion.

Logically, a definition (`has-definition`) is a bi-directional implication, while a normal assertion (`has`) is a unidirectional implication. Consider the two propositions:

$$\forall x\ \mathit{isa}(x, \mathit{Mexican}) \rightarrow \mathit{isa}(x, \mathit{Person}) \wedge \mathit{lives\text{-}in}(x, \mathit{Mexico})$$
$$\forall x\ \mathit{isa}(x, \mathit{Mexican}) \leftrightarrow \mathit{isa}(x, \mathit{Person}) \wedge \mathit{lives\text{-}in}(x, \mathit{Mexico})$$

The first is an assertion ("Mexicans live in Mexico"), but the second is a definition ("Mexicans live in Mexico, *and* all people who live in Mexico are Mexicans."). The second allows KM to recognize instances which satisfy the definition as members of a class. These alternatives would be stated in KM as follows:

```
;;; "Mexicans are people who live in Mexico."
;;; ∀x isa(x, Mexican) → isa(x, Person) ∧ lives-in(x, Mexico)
KM> (Mexican has (superclasses (Person)))
KM> (every Mexican has (lives-in (*Mexico)))

;;; "Mexicans are people who live in Mexico, and vice versa."
∀x isa(x, Mexican) ↔ isa(x, Person) ∧ lives-in(x, Mexico)
KM> (every Mexican has-definition
      (instance-of (Person))
      (lives-in (*Mexico)))
```

17.2 Defined Classes (Testing for Membership)

17.2.1 Example

The below gives an example of the definition of a square:

```
KM> (Square has
        (superclasses (Rectangle)))     ; immediate superclass

;;; “Any shape whose length = its width is a square.”
;;; ∀s isa(s,Square) ↔ isa(s,Shape) ∧ width(s,w) ∧ length(s,l) ∧ w = l[20]
KM> (every Square has-definition        ; ‘Definitional’ properties
          (instance-of (Shape))         ; most general superclass
          (width ((the length of Self)))
          (length ((the width of Self))))

;;; “Squares are pretty.”
;;; ∀s isa(s,Square) → appearance(s,*Pretty)
KM> (every Square has                   ; ‘Incidental’ properties
          (appearance (*Pretty)))
```

Note that being pretty is not part of the definition of a square (no all pretty things are squares); rather it is an implied property (“*if* an object is a square *then* it is pretty”). Note also the distinction between the square’s ‘most general’ class (`Shape`), specified in the definition, and the immediate class(es) (`Rectangle`), specified in the `superclasses` slot. We say `Square` is a *defined subclass* of `Shape`, meaning, in terms of inference, that any created/modified instance of `Shape` will be (re-)tested for `Square` membership. Finally to complete the representation we also need:

```
;;; “All rectangles are shapes.”
KM> (Rectangle has (superclasses (Shape)))
```

Now KM will classify shapes satisfying this definition as squares:

```
;;; “What is the appearance of a rectangle, with width and length 10?”
;;; ∃r isa(r,Rectangle) ∧ width(r,10) ∧ length(r,10) ∧ ANS = { a | appearance(r,a) }
KM> (the appearance of (a Rectangle with (width (10)) (length (10))))
(Changing _Rectangle319’s classes from (Rectangle) to (Square))
(*Pretty)
```

Any time an instance is created or modified, KM automatically attempts to reclassify it by climbing up the isa hierarchy, looking for defined subclasses of the classes which are encountered, and then seeing if that instance satisfies any of those subclasses’ definitions. For example, if a `Man` is defined as a `Person` whose gender is `Male`, then `Man` is a defined subclass of `Person`, and that definition will be tested any time an instance of person is created or modified. Thus, defined subclasses of very general classes (eg. `Red-thing` is a `Thing` which is `Red`) will be tested very frequently (here, every time an instance of `Thing` is created or modified). For efficiency it is probably wise to use such very general classes sparingly in definitions.

---

[20] Strictly, the semantics of the KM expression is that “all the widths = all the lengths”. The above logical expression simplifies this on the assumption that all shapes have exactly one length and one width.

17.2.2 Usage

A primary role of defined concepts is to enable “automatic recognition” of instances of a particular class: As more information becomes known about an instance, it may trigger re-classification of that instance if it satisfies a definition, thus providing new information about that instance. This relaxes the requirement that instances always be manually placed in the most specific, appropriate position in the ontology, as KM will automatically perform this task if an instance satisfies a defined class.

A second useful role is for referring to *all* values of a slot (This can also be done in an alternative way using constraints, Section 12.1, but for now we describe a classification approach). For example, suppose we want to say that *all* wheels of a Nissan cost $30. Rather than annotating wheels on the `Nissan` frame, which would affect only specific instances of wheel, the knowledge engineer can reify the concept of a Nissan wheel:

```
(Nissan-Wheel has
  (superclasses (Wheel)))

;;; “Nissan wheels are wheels on a Nissan (car).”
;;; ∀w isa(w, Nissan-Wheel) ↔ isa(w, Wheel) ∧ ∃n isa(n, Nissan) ∧ parts-of(w, n)
(every Nissan-Wheel has-definition
  (instance-of (Wheel))
  (parts-of ((a Nissan))))                      ; definitional property

;;; “Nissan wheels cost $30.”
;;; ∀w isa(w, Nissan-Wheel) → cost(w, 30)
(every Nissan-Wheel has
  (cost (30)))                                 ; incidental (implied) property

;;; “Every Nissan has four wheels.”
(every Nissan has
  (parts ((a Wheel) (a Wheel) (a Wheel) (a Wheel))))

(Nissan has (superclasses (Car)))
```

Now if we ask for the wheels of a Nissan, they will automatically be classified as `Nissan-Wheel`, thus acquire the property that each costs $30:

```
KM> (the parts of (a Nissan))
(_Wheel242 satisfies definition of Nissan-Wheel:
 Changing _Wheel242's classes from (Wheel) to (Nissan-Wheel))
(... etc. for the other wheels also)
(_Wheel242 _Wheel243 _Wheel244 _Wheel245)

KM> (showme _Wheel242)
(_Wheel242 has
   (cost (30))
   (parts-of (_Nissan241))
   (instance-of (Nissan-Wheel)))           ; Note KM has classified Wheel242

;;; “How much do the wheels of a Nissan cost?”
;;; ∃n isa(n, Nissan) ∧ ANS = sum({ c | ∃w parts(n, w) ∧ isa(w, Wheel) ∧ cost(w, c) })
KM> (the sum of (the cost of (the Wheel parts of (a Nissan))))
(120)
```

17.2.3 Conditionals or Classification?

In many cases, information about 'special cases' of a class can be represented either using an `if...then...` rule in a slot's value, or alternatively by reifying those 'special cases' into subclasses in their own right, and then letting the classification mechanism automatically determine when an instance of the class is also in one of those subclasses. The knowledge engineer needs to make a design decision as to which is best; often the classification alternative results in a more modular (and hence maintainable) knowledge base.

To illustrate this, consider representing that a person's life expectancy is (say) 75 years, unless he/she smokes in which case it is (say) 70 years. We could represent these alternative cases as a rule on `Person`:

;;; "the life expectancy of a smoker is 70, otherwise it is 75."
;;; $\forall p\, isa(p, Person) \rightarrow (\ smokes(p, *Yes) \rightarrow life\text{-}expectancy(p, 70)\ ;\ life\text{-}expectancy(p, 75)\ )$[21]

```
(every Person has
   (life-expectancy ((if   ((the smokes of Self) = *Yes)
                     then 70
                     else 75))))
```

or alternatively we can reify these cases by define two subclasses, `Smoker` and `Non-smoker`:

;;; "People who smoke are smokers."
;;; $\forall s\ isa(s, Smoker) \leftrightarrow isa(s, Person) \wedge smokes(s, *Yes)$

```
(every Smoker has-definition
   (instance-of (Person))
   (smokes (*Yes)))
```

;;; "A smoker's life-expectancy is 70 years."
;;; $\forall s\ isa(s, Smoker) \rightarrow life\text{-}expectancy(s, 70)$

```
(every Smoker has
   (life-expectancy (70)))
```

;;; "People who don't smoke are non-smokers."
;;; $\forall n\ isa(n, Non\text{-}Smoker) \leftrightarrow isa(n, Person) \wedge smokes(n, *No)$

```
(every Non-Smoker has-definition
   (instance-of (Person))
   (smokes (*No)))
```

;;; "A non-smoker's life-expectancy is 75 years."
;;; $\forall n\ isa(n, Non\text{-}Smoker) \rightarrow life\text{-}expectancy(n, 75)$

```
(every Non-Smoker has
   (life-expectancy (75)))
```

Both alternatives produce the same answer to the query:

```
KM> (the life-expectancy of (a Person with
                                (smokes (*Yes))))
(70)

KM> (the life-expectancy of (a Person with
                                (smokes (*No))))
(75)
```

---

[21] Assuming $smokes()$ and $life\text{-}expectancy()$ are single-valued. $(X \rightarrow Y\ ;\ Z)$ is shorthand for $(X \rightarrow Y) \wedge not(X) \rightarrow Z$, where $not()$ is negation as failure.

However the classification approach, although more verbose here, is more modular – for example we can easily add additional properties about smokers/non-smokers by simply adding information to the `Smoker` or `Non-Smoker` frames. The alternative with `if...then...` rules would be to add more conditional expressions on the `Person` frame, essentially "mixing up" information about people, smokers and non-smokers in a single frame. This will rapidly produce a large and messy frame structure if a lot of information is to be included. The knowledge engineer thus needs to carefully decide whether if...then... rules should be used, or whether the objects satisfying the rules should be reified into new subclasses with definitional properties.

17.2.4 Definitions based on Constraints rather than Slot-Values

Some definitions are based on constraints, rather than the presence of particular slot-values. For example, how can we express the definition that "an adult is a person over 21 (say) years old"? To do this, the constraint "age $\geq$ 21" needs to be re-expressed as a slot-value test, by introducing the property `over-21` (say), and then testing that this property holds:

```
;;; "A person is 'over 21' if their age is greater than or equal to 21."
;;; ∀p isa(p, Person) → ( (∃a age(p, a) ∧ a ≥ 21) → over-21(p, *Yes) )
KM> (every Person has
        (over-21 ((if ((the age of Self) >= 21) then *Yes))))

;;; "Adults are people over 21, and vice versa."
;;; ∀a isa(a, Adult) ↔ isa(a, Person) ∧ over-21(a, *Yes)
KM> (every Adult has-definition
        (instance-of (Person))
        (over-21 (*Yes)))

;;; Create a 30 year old person...
KM> (a Person with (age (30)))
(_Person0 satisfies definition of Adult:                 ; KM correctly concludes
 Changing _Person0's classes from (Person) to (Adult))   ; he/she is an adult
(_Person0)

;;; "Is this person an adult?"
KM> (_Person0 isa Adult)
(t)                                                      ; Yes!
```

17.2.5 Classification and the Closed-World Assumption

What happens if a definition refers to a slot, whose value is unknown? For example:

```
;;; "If a car is the favorite color of it's owner, then it's a nice car."
KM> (every Nice-Car has-definition
       (instance-of (Car))
       (color ((the favorite-color of (the owner of Self)))))
```

If we now create an instance of a `Car` (whose owner is unspecified), KM would classify this as a `Nice-Car`:

```
KM> (a Car)
CLASSIFY: _Car23 is a Nice-Car!
```

This would happen as, when testing if `_Car23` satisfies the definition of `Nice-Car`, KM compares `_Car23`'s color (NIL), with the favorite color of the owner of the car (also NIL), and as they are the same, considers the definition satisfied. This is undesirable, as the author clearly intended the definition to only apply when the owner's favorite color was known, i.e., the author did not intend the closed-world assumption (CWA) to apply.

In fact, KM has a tweak in to prevent this kind of inference: Every definitional expression is expected to return at least one value, and if there are more definitional expressions than slot values on the instance being tested, the classification *fails*. In other words, KM assumes that each definitional expression was put there for a reason. Note that this does not include constraint expressions in definitions, and so care needs to be taken still with definitions based on constraints.

If KM makes a classification using the closed-world assumption, and new information comes in invalidating that classification, KM will *not* be able to recognize this and undo that classification.

17.3 Defined Instances (Testing for Equivalence)

As well as defining properties for class membership, the user can also define properties for *equivalence* – that is, if an anonymous instance I satisfies the defining properties of instance DI, then KM will conclude that I *is*, in fact, DI (ie. is coreferential with DI). For example, KM may recognize that a country being described is the UK, or that a car being described is in fact Joe's car. This classification machinery is similar to that for recognizing class membership, except the final step involves unifying (rather than defining instance-ship between) the instance being tested and the defined object. This unification is performed using the `&` unification operator, as described in Section 10. For example:

```
;;; "London is that city which is the capital of the UK"
;;; ∀isa(l, *City) ∧ capital-of(l, *UK) ↔ l = *London
KM> (*London has-definition
       (instance-of (City))              ; most general class
       (capital-of (*UK)))

;;; "London is a big city, with a population of ten million."
;;; isa(*London, Big-City) ∧ population(*London, 10000000)
KM> (*London has
      (instance-of (Big-City))           ; immediate class
      (population (10000000)))

KM> (Big-City has (superclasses (City)))

;;; "How big is a (the) city which is capital of the UK?"
;;; ∃c isa(c, City) ∧ capital-of(c, *UK) ∧ ANS = { p | population(c, p) }
KM> (the population of (a City with (capital-of (*UK))))
(_City24 satisfies definition of *London: Unifying _City24 with *London)
(10000000)
```

Note that KM considers named instances (those whose names do not begin with a '_') as distinct, and does not allow them to unify (Section 10). Thus if an anonymous instance satisfies more than one (named) instance's definition, KM considers this a KB error and will report it to the user.

Historical Note: By default, if *any* of an instance's slot values change, KM will re-attempt classification of that instance. This includes testing definitions which do not directly reference that slot, as the change may indirectly affect the slots which *are* referenced by the definition ("indirect classification"). In addition, computations during classification may themselves trigger additional classifications ("recursive classification"). In older versions of KM, these default behaviors could (and still can) be turned off by setting the global variables `*indirect-classification*` and `*automatic-classification*` to nil. This improves reasoning efficiency but at the cost of adding incompleteness to KM's reasoning, and is not recommended.

There is still a remaining source of incompleteness in KM's classification, namely when changes *elsewhere* in the KB imply an instance's classification should change, i.e., non-local effects do not trigger classification of an instance. This is described later in Section 30.1.2.

18 Intensional Representations and Subsumption

18.1 Introduction

Closely related to automatic classification are the topics of intensional representations and subsumption. While automatic classification involves writing class definitions and testing whether instances satisfy them, intensional representations concern manipulating definitions as objects in their own right, and subsumption involves ascertaining whether one definition is "more general" than another (i.e., a class-class test, in contrast to automatic classification's class-instance tests).

As background, it is important to distinguish between a concept's *intension* and *extension*. The *extension* of a concept (class) is the set of its members (under some interpretation), while the concept's *intension* is the concept itself. For example, the intension of the concept of dogs might be represented by the symbol `Dog` (a class), while the extension of the concept of dogs would be the set of all instances of the class `Dog`. Note that intensions and extensions are related but distinct: two concepts (eg. "the morning star" and "the evening star", or "unicorns" and "dragons") may have the same extension but still be distinct (have different intensions). In KM, intensions are denoted by classes, and extensions are denoted by those classes' instances. Formally, a concept's extension is a mapping from individuals to truth values (under some interpretation), while a concept's intension is a mapping from possible worlds onto extensions.

Intensional representations are important, because in many cases it is useful to manipulate concepts directly rather than manipulate just their instances. In particular, natural language uses intensional descriptions extensively, e.g., "John wants to play soccer" refers to a type (class) of activity (namely, playing soccer), rather than a specific instance of that activity (e.g., `_Play203`). Intensional representations provide a means for representing these types in a structured form, and exploiting that structure in reasoning (compared with, say, just using an unstructured class name like `PlaySoccerEvent`).

18.2 Class Descriptions

In the simplest case, we denote a concept (class) using a symbol, e.g., `Dog` denotes the concept of dogs. As a generalization of this, we can also denote classes using a *structure*, of the form:

(`the-class` *class [*`with` *slotsvals]*)

For example, the expression:

```
KM> (the-class Cat with
       (color (*Black)))
```

denotes the class "black cats". Unlike other expressions, KM *does not evaluate* `the-class` structures. Rather, they behave like "quoted expressions", to be manipulated by KM just like atomic class names. Note the following important equivalence holds:

`Cat` = `(the-class Cat)` = a denotation of the concept "cat"

The properties specified in the `the-class` expression are the *defining properties* of this concept, and thus `the-class` expressions and automatic classification (Section 17) are closely related (including sharing some subroutines in the software implementation). The main difference is that a defined class is a named (reified) class which is a permanent part of the KB taxonomy, while a `the-class` expression is a transient, non-reified concept. The ability to denote classes by structures as well as symbols is important: it means we do not have to reify (give an explicit name to) every concept of interest, which can otherwise clutter a KB, and instead create class descriptions compositionally on demand when they are needed for specific tasks. For example, in the domain of chemistry, if we want to test whether a chemical instance is a yellow precipitate, we could use a class description rather than multiple attribute tests:

```
KM> (_Chemical1 isa (the-class Precipitate with
                         (color (*Yellow))))
```

Similarly, we might choose to represent the primary purpose of an artifact by an event class, e.g., "the purpose of a car is to transport people", which could be encoded:

```
KM> (every Car has
      (purpose ((the-class Transport with
                   (transporter (Self))
                   (transportee ((a Person)))))))
```

Note that the purpose is not a single, specific transportation event; rather, the purpose is characterized by an event *type* (class) which it was designed to perform. This can be captured parsimoniously using a `the-class` expression, as shown above, rather than reifying a class named `Transportation-Of-People-By-Car` or something similar.

18.3 Subsumption

It is often useful to know when one concept description is "more general" than another, ie. "subsumes" another. For example `(the-class Car)` subsumes `(the-class Car with (color (*Red)))`, as all members of the latter are necessarily also members of the former. Formally, concept (class) C_1 subsumes concept C_2 if "being a C_2" logically implies "being a C_1", that is, if $\forall x \; isa(x, C_2) \rightarrow isa(x, C_1)$. We can equivalently define subsumption using denotational semantics: If $\mathcal{C}_1^I$, $\mathcal{C}_2^I$ are the extensions of the sets $\{x|isa(x, C_1)\}$, $\{x|isa(x, C_2)\}$ respectively under some interpretation I, then C_1 subsumes C_2 if $\mathcal{C}_1^I \supseteq \mathcal{C}_2^I$ for all interpretations I. This is the standard definition of subsumption used in Description Logics.

KM's comparison operator `subsumes` will compute subsumption. This involves testing that the subsumer's class is a superclass of the subsumee's, and testing that the subsumee necessarily has all the properties of the subsumer. These tests may involve inferencing, but note that the subsumption test is not logically complete (which in general is intractable), as KM underlying inference method (backward-chaining, negation-as-failure) is incomplete (see Section 30).

For example, given the below query:

```
;;; "Are all elephants animals with trunks?"
KM> ((the-class Animal with (parts ((a Trunk)))) subsumes (the-class Elephant))
```

KM will check that

(i) `Elephant` is a subclass of `Animal`

(ii) All Elephants have (at least) a trunk part. (This may involve inferencing).

As `(the-class Elephant)` is equivalent to `Elephant`, we could have asked equivalently:

```
;;; "Are all elephants animals with trunks?"
KM> ((the-class Animal with (parts ((a Trunk)))) subsumes Elephant)
```

The inverse operator `is-subsumed-by` is also defined in KM.

18.4 `isa`

The operator `isa`, described earlier, checks whether an instance is in a class (rather than `subsumes`, which compares a class with a class). With `isa`, *class* can be an intensional class description, not just a class name, e.g.:

```
KM> (a Dog with (color (*Black)) (age (20)))
(_Dog1)

KM> (_Dog1 isa (the-class Dog with (color (*Black))))
(t)
```

`subsumes` and `isa` can be thought of as the intensional equivalents of `is-superset-of` and `includes` operators (Section 13), which manipulate sets.

Historical Note: The operators `covers` and `is-covered-by` in earlier versions of KM have been replaced by this generalized version of `isa`, and are no longer supported.

19 Explanations

19.1 Facts and Rules in KM

KM includes a method for explaining (justifying) the facts that it concludes through inference. A fact is an instance, slot, and value for that slot – in logic, this is simply a ground, binary assertion $slot(instance, value)$. To refer to facts (i.e., treat them as objects in their own right), KM uses a "triple" notation of the form:

;;; $slot(instance, value)$
(`:triple` *instance slot value*)

This ability to manipulate ground propositions as objects in their own right is used extensively in KM's situation mechanism [8]. It is also used here for explanation.

Each expression on a frame's slot can be thought of as a "rule" (for computing that slot's values). An explanation for a fact in KM constitutes an (English rendition of) the rule(s) which concluded that fact, plus, recursively, explanations for facts supporting those rules. For example, given the KB:

```
;;; Every car has an engine.
KM> (every Car has
      (parts ((a Engine) (a Chassis))))
```

and the queries:

```
KM> (a Car)
(_Car0)

KM> (the parts of _Car0)
(_Engine1 _Chassis2)
```

Then the concluded fact

```
(:triple _Car0 parts _Engine1)
```

is supported by the "rule"

```
(every Car has (parts ((a Engine))))
```

19.2 The (`why`) Command

During reasoning, for every ground fact concluded, KM records the rule(s) (KM expressions) which were used to derive it. This produces a database of "proof tree fragments" (the explanation database), from which a full explanation can be generated. To see entries in this database, showing which rule(s) support which facts, use the command (`why` *[triple]*). For example:

```
KM> (why (:triple _Car1 parts _Engine2))
(:triple _Car1 parts _Engine2 [in *Global]) because:
   RULE: (every Car has (parts ((a Engine))))
```

If the *[triple]* argument is omitted, `(why)` will default to showing rules answering the previous query. In addition, if entry and exit paraphrases for this rule are provided (using a `comment` assertion, described in the next subsection), these will also be shown in the output.

19.3 Using English Paraphrases of Rules

19.3.1 Comment Tags

To generate explanations in English, the user can attach English paraphrases, or "comments" to individual rules. Comments are expressed using a structure:

(`comment` *tag [exit-text entry-text supporting-facts]*)

In fact, the comment structure includes two paraphases of the rule, one to print when the rule is applied (the *entry-text*), and one to print when the rule's computation is complete (the *exit-text*). We illustrate these shortly. *tag* is a symbol enclosed in square brackets `[]`. To associate this comment with a KM expression, the same tag is placed (anywhere) within that KM expression. (These tags are filtered out before expression itself is evaluated). For example:

```
KM> (every Car has
       (parts ((a Engine [Car1]))))

;;; The explanation text attached to this rule
KM> (comment [Car1] "Thus, this car has an engine." "All cars have engines.")
KM> (a Car)
(_Car2)

KM> (the parts of _Car2)
(_Engine3)
```

To see an explanation expressed using these English paraphrases, use the command `(justify` *[triple]*`)`. For example:

```
KM> (justify (:triple _Car2 parts _Engine3))
All cars have engines.
Thus, this car has an engine.
```

19.3.2 Text Generation

The full power of KM's text generation can be used in the entry and exit text parts of comment tags. Thus these text paraphrases can be strings, `:seq` text sequences, or arbitrary KM expressions which evaluate to a string or text sequence. Details on text generation in KM is given in Section 16.

19.4 Recursive Explanations

The comment tag optionally includes a last argument, *supporting-facts*, in which the user can explicitly list the "key" facts which that rule depends on. These facts are expressed either as one or a set of `:triple` structures, and KM's explanation mechanism will recursively explain these facts, when explaining the main rules. The idea here is that the user explicitly lists those facts which caused the rule to succeed, and which are worth further explanation. This is for two purposes: First, KM currently cannot work out the supporting facts automatically, requiring the user to specify them. Second, KM does not know which supporting facts are "interesting" to explain and which are obvious/uninteresting; the ones the user lists are treated as "interesting", allowing the user substantial control over the depth and direction of the final explanation.

The following illustrates this. Note the supporting fact attached to the comment `[Tax1]`. A wildcard `*` can be used in the third argument of a triple (as shown below), so that all values for this argument place are explained.

```
KM> (every Car has
      (cost (((the pretax-cost of Self) * (1 + (the tax-rate of Self)) [Tax1])))
      (pretax-cost (((the base-cost of Self) + (the options-cost of Self) [PreTax1]))))

KM> (comment [Tax1]
        (:seq "So the total cost of the car is" (the pretax-cost of Self)
              "* (1 + " (the tax-rate of Self) ") = " (the cost of Self) ".")
        "A car's total cost is its pretax cost plus tax."
        (:triple Self pretax-cost *))

KM> (comment [PreTax1]
         (:seq "So, the car costs (pretax) " (the base-cost of Self) "+"
               (the options-cost of Self) "=" (the pretax-cost of Self) ".")
         "A car's pretax cost is its base cost + options cost.")

KM> (*MyCar has
      (instance-of (Car))
      (base-cost (10000))
      (options-cost (2000))
      (tax-rate (0.08)))

KM> (the cost of *MyCar)
(12960.001)
```

```
KM> (justify)
I'll assume you're asking me to justify:
   (the cost of *MyCar) = (12960.001)...

A car's total cost is its pretax cost plus tax.
  A car's pretax cost is its base cost + options cost.
  So, the car costs (pretax) 10000 + 2000 = 12000.00.
So the total cost of the car is 12000 * (1 + 0.0800) = 12960.00.
```

Note the supporting facts listed for comment `[Tax1]` only includes the pretax-cost, and not the tax-rate. If we had wanted to include both, the comment would have looked instead:

```
KM> (comment [Tax1]
        (:seq "So the total cost of the car is" (the pretax-cost of Self)
              "* (1 + " (the tax-rate of Self) ") = " (the cost of Self) ".")
        "A car's total cost is its pretax cost plus tax."
        (:set (:triple Self pretax-cost *)
              (:triple Self tax-rate *)))
```

and we would have had to attach a comment tag to facts/rules which computed the tax rate.

Using this mechanism, sophisticated explanations can be constructed automatically by KM. Some examples from the Halo project can be seen at http://www.projecthalo.com/ (see the Results Browser, best viewed with Internet Explorer).

20 Tagging Instances

20.1 Removing Ambiguity in Paths

A KM path allows the user to reference a particular instance by its location in the KB. However, in some cases, in particular when a slot has multiple instances of the same type, the reference can be ambiguous. The tagging mechanism is designed to address this, by allowing the user to "tag" particular instances with labels, and then use those labels to help identify instances.

For example, consider the following script (simplified by omiting information on who the actors are):

```
KM> (every Get-Coffee-ToGo has
       (subevents (
         (a Enter)              ; enter the store
         (a Walk)               ; walk to counter
         (a Purchase)           ; purchase coffee
         (a Walk)               ; walk back to the door
         (a Exit))))            ; leave
```

Note there are two `Walk` subevents. Suppose we wished to refer to just the first of these. The expression `(the Walk subevents of Self)` refers to both `Walk` subevents, and so is inadequate. While we could write a more complex expression which would uniquely identify the first walk subevent, a simpler solution is simply to "tag" the two subevents with different labels using the form:

`(a` *expr* `called` *tag* [`with` *slotsvals*])

as illustrated below:

```
KM> (every Get-Coffee-ToGo has
       (subevents (
         (a Enter)                              ; enter store
         (a Walk called "walk to counter")      ; walk to counter
         (a Purchase)                           ; purchase coffee
         (a Walk called "walk back out")        ; walk back to the door
         (a Exit))))                            ; leave
```

Having done this, we can then select instances with specific tags using the form:

(*expr* `called` *tag*)

To evaluate this, first *expr* is evaluated and then only the instance(s) in the result which are tagged with *tag* are returned. Thus, this expression acts as a filter. In the example, we would thus set up next-event links like this:

```
KM> (every Get-Coffee-ToGo has
       (subevents (
         (a Enter with
           (next-event (((the Walk subevents of Self) called "walk to counter"))))
         (a Walk called "walk to counter" with
           (next-event ((the Purchase subevents of Self))))
         (a Purchase with
           (next-event (((the Walk subevents of Self) called "walk back out"))))
         (a Walk called "walk back out" with
           (next-event ((the Exit subevents of Self))))
         (a Exit))))
```

Tags can be any atom (strings, numbers, or symbols). They can also be negated, e.g., `(<>` *tag*`)` (see next Section). Tags are implemented simply by placing the tags on a built-in slot named '`called`'.

20.2 Controlling Unification

Tags have a second purpose also, namely to allow the user to help guide unification. When two sets of values are unified, there may be ambiguity about which elements should unify with which. Normally, KM tries to follow a pairwise left-to-right strategy. However, if an element from each set happens to have the same tag, then KM will unify those two elements first, in preference to the default ordering constraints.

Similarly, tags can be used to block unification in two ways. First, recall that with `called` tags, an instance may have more than one tag assigned to it. Now, KM provides a similar operator named `uniquely-called`, which behaves just like `called` except that an instance may have at most one uniquely-called tag. Thus, if two instances have different uniquely-called tags, then they cannot be unified (as this would violate the at-most-one-value constraint). Thus by placing uniquely-called tags on instances, the user can explicitly declare that two instances are necessarily distinct:

```
KM> (a Dog uniquely-called "fido")
(_Dog6 #|"fido"|#)

KM> (a Dog uniquely-called "rover")
(_Dog7 #|"rover"|#)
```

```
(1 inferences and 28 KB accesses in 0.0 sec)

KM> (_Dog6 & _Dog7)
NIL
```

Like `called` tags, uniquely-called tags are implemented by placing them on a built-in slot named `uniquely-called`. Unlike `called`, `uniquely-called` is a single-valued slot (i.e., cardinality N-to-1).

Note also that as a user interface feature, tags are printed out in comments `#|...|#` after the instance names to help the user.

Objects can be both called and uniquely-called things at the same time. If an object is uniquely-called something, then it is also called that something.

Second, as mentioned earlier, tags can also be negated, expressing that an instance is *not* called a particular tag. Again, this can block unification, for example:

```
;;; A dog not called Fido
KM> (a Dog called (<> "fido"))
(_Dog8)

;;; Can that dog be unified with a dog called Fido?
KM> (_Dog8 &? (a Dog called "fido"))
NIL                                     ; No
```

21 Defaults

KM treats rules in the KB as "always true", and will report an error if a rule is contradicted by data. In contrast, people often express common-sense knowledge using rules which are somewhat weaker than this, where there may be valid exceptions (e.g., "Birds can fly.", "Cars have four wheels."). KM provides two simple mechanisms for expressing such rules, which we now describe. Note that these mechanisms do not provide support for non-monotonic reasoning (the ability to retract old conclusions as new facts are added), they merely allow knowledge to be expressed in a layered form (rules + exceptions). That is, KM is assuming full knowledge at the time conclusions are being drawn, and that they will not need to be retracted later.

21.1 Overriding Inheritance

Normally, an instance inherits from *all* of its classes; that is, when a slot is queried, KM ascends the taxonomy from that instance, collecting all expressions it encounters along the way, then evaluates and combines them (Section 10). However, KM also provides an "inherit with overrides" mechanism, which allows this inheritance behavior to be changed for certain slots. With this mechanism, rather than inheriting rules from all classes, an instance will only inherit rules from the *first* class KM encounters which has rules on the target slot, when ascending the taxonomy. With multiple inheritance, all upward branches are explored, each exploration stopping when some rules are encountered on the slot of interest (ie. KM gathers all rules from all the most specific classes with rules for that slot on them). This behavior is slot-specific: To switch on this behavior for a slot, the slot is declared as an "inherit with overrides" slot by setting its `inherit-with-overrides` property to `t`, e.g.

```
KM> (text-description has
      (instance-of (Slot))
      (inherit-with-overrides (t)))
```

As a result, in this example, only the lowest rules on the parent classes will be used to compute the `text-description` of an instance.

21.2 `:defaults`

A second, related mechanism in KM is to declare particular expressions as "default" by wrapping them in a `(:default` *expr*`)` form. Such defaults can be overriden by constraints without generating an error. For example:

```
;;; All cars have an engine and (by default) a seat
KM> (every Car has
       (parts ((a Engine) (:default (a Seat)))))

;;; My car doesn't have a seat
KM> (*MyCar has
       (instance-of (Car))
       (parts ((mustnt-be-a Seat))))

;;; What are the parts of my car?
KM> (the parts of *MyCar)
(_Engine2)                              ; note, no Seat
```

This default mechanism is fairly simple-minded at present:

- Defaults can be only overriden by constraints.
- If two default values conflict (and the slot is single-valued), KM will report an error.
- There is no support for "default constraints" in KM, i.e., values cannot override constraints.
- There is no non-monotonicity implemented: If a default is applied, and later information contradicts it, Km will not drop the default, but instead report an error. This is an implementation limitation.

22 Prototypes

22.1 Introduction

Prototypes are a new and significantly different style of knowledge representation in KM. A prototype is an *example* of a class member, and prototype facts are transferred to other class members by (for each member) *cloning* the prototype and *unifying* that clone with the member. Prototypes offers some important advantages:

1. By representing facts about a class using a concrete example, prototypes allow easy visualization and editing of those facts. This has been fundamental in a system called Shaken, a KM-based knowledge acquisition system which uses graphical techniques to display and enter knowledge [15].

2. KM provides a special interface (KM's prototype mode, described below) for encoding prototypes which has some commonalities with a natural language (NL) interface. In particular, it allows coreference to be expressed using descriptions (e.g., `(the Car)`), rather than long paths (e.g., `(the car of (the spouse of Self))`). This provides a potential bridge for connecting with NLP technologies.

3. Finally, it is not necessary that a prototype contain *all* the information about a class; rather, a class can have multiple prototypes, each expressing different subsets of information about a class. This potentially allows class knowledge to be factored into separate pieces.

There are also some disadvantages with prototypes: They can be computationally slower due to their extensive use of unification, KM's NLP-like interface for easy encoding of prototypes has limited expressivity (although prototypes can be encoded directly in KM with full expressivity), and KM's explanation capabilities do not currently extend to knowledge encoded in prototypes.

The word "prototype" in the AI literature has an additional connotation, namely that the facts about it are only "typical", and thus can be overridden or ignored if they clash with what is known about a particular example. This connotation does *not* apply to KM prototypes; all prototype facts in KM are treated as universally true[22], and an error will be reported if they clash with what is known about an instance. In other words, KM prototypes are a clone-and-unify mechanism for deductive reasoning, not a mechanism for handling typicality.

22.2 KM's Prototype Mode

22.2.1 Introduction

There are two ways of entering prototypes: using KM's friendly prototype mode, or directly in "raw" KM. We first describe the prototype mode. By using the prototype mode, the user does not need to know all the implementation details of prototypes; however, the mode is also limited in the kind of axioms which can be entered for prototypes. Conversely, by entering prototypes in "raw" KM, the user has full expressivity but needs to understand more about KM's internal mechanism for dealing with prototypes.

Prototypes entered in prototype mode are converted into "raw" KM, and the user can see the result of this conversion simply by printing out the KB (using `save-kb` or `write-kb`).

22.2.2 Details

A prototype is an explicit example of a class member, and is denoted by a special kind of instance called a proto-instance. One can view a proto-instance as not being a "real" instance as it does not denote any specific individual in the world, although this distinction is mainly in the eye of the user. For KM-generated proto-instances, KM gives them names starting with "`_Proto`" rather than just "`_`" (the prefix is purely cosmetic, and has no functional significance). The prototype description itself may involve other proto-instances also, e.g., a prototype car might have an engine part, represented as a proto-instance of car having a proto-instance of engine on its "parts" slot. In this example, we call the proto-instance of car the *prototype root*, and the proto-instances of car and engine (and any other involved instances) are the *prototype participants*.

The idea with KM's prototype mode is to allow the user to gradually build up this "network" of proto-instances, describing the prototype, by repeatedly creating and relating instances together. To define a new prototype, the command `(a-prototype` *class*`)` is used. This command creates a proto-instance of *class*, and also causes KM to enter prototype mode, denoted by the prompt `[prototype-mode] KM>`. This mode has three important characteristics:

1. Inheritance is turned off, and the entire KB is invisible except for the taxonomy (instance-of and superclass relationships), and anything created within that prototype mode.

---

[22] bar explicit use of `:default` statements on prototypes

2. All KM expressions are fully and immediately evaluated, i.e., paths are not preserved, but rather their referents are immediately located. This allows the user to refer to earlier created instances using paths and definite descriptions (see Section 15).
3. When the user enters prototype mode, the list of instances created (the "object stack", Section 15) is cleared. Then, as the user creates instances within the prototype mode, they are added to the stack, allowing the user to refer back to those instances using definite descriptions. For example, `(the Car)` will find the instance of car created earlier within the prototype environment. When the user exits prototype mode, using the command `(end-prototype)`, all the instances on the stack are noted as part of (participants in) the prototype being defined.

For example, to define a simple prototype car, we could write:

```
KM> (a-prototype Car)                                          ; [1]
(_ProtoCar1)

[prototype-mode] KM> ((the Car) has                            ; [2]
                          (parts ((a Engine)
                                  (a Fuel-Tank))))
(_ProtoCar1)

[prototype-mode] KM> ((the Fuel-Tank) has                      ; [3]
                          (connected-to ((the Engine))))
(_ProtoFuel-Tank3)

[prototype-mode] KM> (show-obj-stack)                          ; [4]
   _ProtoFuel-Tank3
   _ProtoEngine2
   _ProtoCar1

[prototype-mode] KM> (end-prototype)                           ; [5]

KM>
```

Statement `[1]` causes entry into the prototype environment, and creates a proto-instance of `Car`. This instance is special, as it is the 'root' of the prototype, and corresponds to the class of objects which the prototype is universally quantified over. Statement `[2]` refers back to this car, and attaches two parts to it. Note that we can refer back to `(the Car)`, as the car is a concrete (proto-)instance, and attach new properties to it. In statement `[3]` we again refer back to the tank and engine, created earlier in statement `[2]`. Statement `[4]` displays the prototype instances created (the prototype participants). The prototype is now defined, represented by these three frames in the KB.

22.2.3 Inference with Prototypes

KM uses prototypes for inference as follows: If ever KM is trying to find the slot-value of an instance I, and I is in the same class as the root of a prototype (eg. `Car`, above), and that prototype includes information about that slot, *then* the prototype graph is cloned, and the clone unified with instance I. As a result, I will acquire the properties the prototype described. For example:

```
KM> (a Car)
(_Car4)

KM> (the parts of _Car4)
(COMMENT: Cloned _ProtoCar1   -> _Car5 to find (the parts of _Car4))
(COMMENT: (_Car4 & _Car5) unified to be _Car4)
(_Engine6 _Fuel-Tank7)

KM> (the connected-to of _Fuel-Tank7)
(_Engine6)
```

In this example, a copy (clone) of `_ProtoCar1` and all its attached proto-instances was created (`_Car5`), which was then unified with the original car (`_Car4`), thus supplying information about the parts of the car. As a result, the query could be answered.

The user can also explicitly clone a prototype using the command (`clone` *protoinstance*):

```
KM> (clone _ProtoCar1)
(_Car8)
```

This command can be issued from within a prototype environment, if the user wants to import a (clone of) one prototype into the description of another.

22.2.4 Prototypes with Definitional Properties

(`a-prototype` *class*) has a more general form, namely (`a-prototype` *class* `with` *slotsvals*). The values in this expression are treated as the definitional properties of this prototype. We can thus, for example, define a prototype of 'a car with a radio":

```
KM> (a-prototype Car with
      (parts ((a Radio))))
```

All these properties constitute the 'definitional properties' of the prototype, also called the "prototype scope" (next Section). For the prototype to apply to a specific instance, the instance must have all these defining properties, i.e., be within the prototype's scope.

22.3 Defining Prototypes Directly in KM

As the user types in commands in prototype mode, KM synthesizes an internal representation of the prototype. Thus, an alternative to using the prototype mode is to simply enter the internal representation directly, e.g., in a .km file. This requires a bit more understanding of this internal representation, but also allows the prototype to include paths, rules, constraints, or any other form as desired.

The internal form is best described by example. If we were to save the KB to a file after the earlier example (Section 22.2.2), the parts defining the prototype would look:

```
(_ProtoCar1 has
  (instance-of (Car))
  (prototype-of (Car))
  (prototype-scope (Car))
  (prototype-participants (_ProtoCar1 _ProtoEngine2 _ProtoFuel-Tank3))
  (parts (_ProtoEngine2 _ProtoFuel-Tank3)))
```

```
(_ProtoEngine2 has
  (instance-of (Engine))
  (part-of (_ProtoCar1))
  (connected-to (_ProtoFuel-Tank3)))

(_ProtoFuel-Tank3 has
  (instance-of (Fuel-Tank))
  (part-of (_ProtoCar1))
  (connected-to (_ProtoEngine2)))
```

There are three key, built-in slots on the prototype root (`_ProtoCar1`) which define this little "graph of instances" as a prototype:

`prototype-of` This slot gives `_ProtoCar1` the special status of being a prototype, rather than just a normal instance, i.e., available for cloning. (One can thus stop it being a prototype simply by deleting this slot).[23]

`prototype-scope` This slot defines the scope of the prototype, i.e., this prototype applies to any instance of its scope (`Car`, in this case).

`prototype-participants` This slot defines what the actual elements of the prototype are, i.e., the extent of the prototype. Cloning the prototype means cloning all the frames listed here, including all the slot-value relationships on those frames.

The difference between `prototype-of` and `prototype-scope` is implementational. Any instance which is in the `prototype-of` class will be considered by the inference engine as a *potential* target to apply the prototype to during reasoning. Then, only if that instance is within the `prototype-scope` will the prototype actually be applied (i.e., cloned and unified). In this case, the two class values are the same, but for prototypes with additional defining properties, the prototype-of and prototype-scope will differ, as illustrated below:

```
;;; Define a prototype of a car with a radio
KM> (a-prototype Car with (parts ((a Radio))))  ; NB defines the prototype-scope
(_ProtoCar12)

;;; It has a battery, connected to that radio
[prototype-mode] KM> ((the Car) has
                         (parts ((a Battery with
                                     (connected-to ((the Radio)))))))

[prototype-mode] KM> (end-prototype)

KM> (showme _ProtoCar12)
(_ProtoCar12 has
  (instance-of (Car))
  (prototype-of (Car))
  (prototype-scope ((the-class Car with (parts ((a Radio))))))
  (prototype-participants (_ProtoCar12 _ProtoRadio13 _ProtoBattery14))
  (parts (_ProtoRadio13 _ProtoBattery14)))
```

Note in this example the prototype-scope is more specific than the prototype-of. Thus, this prototype will only be applied to instances of the prototype-scope `(the-class Car with (parts ((a Radio))))`, as desired.

---

[23] The inverse slot of `prototype-of` is `prototypes`. Also, the query `(the all-prototypes of` *class*`)`, which uses the special transitive closure slot `all-prototypes`, returns all prototypes of *class* and all its subclasses (Section 4.3)

23 Theories

KM provides a (relatively untested) mechanism allowing the KB to be divided up into different partitions, or *theories*. A KM theory is similar to the notion of a context [16] or microtheory [17], and allows the knowledge engineer to turn "on" or "off" different sets of axioms. Implementationally, it uses some of KM's machinery for reasoning with situations, described in a separate manual [8].

A theory itself is denoted by an instance of the built-in class `Theory`. A theory can be "entered" by the command `(in-theory` *theory*`)`, and when entered, the KM prompt is modified to include the theory name as a prefix. Once the user is "in" the theory, all frames which are entered will be stored in that theory. To "exit" the theory, the directive `(end-theory)` is used.

By default, from the normal KM prompt, the contents of all theories are *invisible* to the reasoner. To make a theory's contents visible, the directive `(see-theory` *theory*`)` is used, and conversely `(hide-theory` *theory*`)` will make the contents invisible again. `(visible-theories)` provides a list of the currently visible theories. The following illustrates the use of theories:

```
;;; Create an instance to denote a new theory
KM> (*MyTheory has (instance-of (Theory)))

KM> (in-theory *MyTheory)
(COMMENT: Changing to Theory *MyTheory)

{*MyTheory} KM> (*Fred has (age (21)))

{*MyTheory} KM> (end-theory)

;;; Computation (contents of *MyTheory not visible)
KM> (the age of *Fred)
NIL

;;; Make *MyTheory visible
KM> (see-theory *MyTheory)

;;; Computation (contents of *MyTheory visible)
KM> (the age of *Fred)
(21)

;;; List currently visible theories
KM> (visible-theories)
(*MyTheory)
```

Implementationally, when a query is issued from the KM prompt, KM looks for relevant frames in the main (global) KB, and all the visible theories. Note that (as always) KM may cache the results of a query and a query's subgoals in the KB, and those cached results will remain even if the user subsequently hides the theories used – in other words, the act of issuing a query may cause information to be transferred from visible theories to the main (global) KB, and that information will remain even if the theory subsequently is hidden. Generally, the contents of the contributing theories themselves will remain unchanged.

Similarly, if a query is issued from *within* a theory (i.e., at a {*theory*} `KM>` prompt, after the user has "entered" a theory), KM will combine frame information from that theory, the main (global) KB, and any other theories which are currently visible. Again, results of evaluating that query and its subgoals may be cached *within that theory*, i.e., there may be a transfer of information from the main (global) KB and other theories into the theory being queried. Generally, only the contents of the queried theory will be affected, i.e., a query from within a theory will not change the contents of the main (global) KB nor any other visible theory. If the user wants to see the result of reasoning with the main (global) KB plus various other theories, without affecting any of them, then he/she could create a new (essentially temporary) theory to reason in, specifically for that purpose.

Although, in general, changes are localized to the theory where a query is issued, there is one important exception to this, namely unification. Unification is a *global* operation, and unification within a theory will cause unification throughout the entire KB (there is no notion of "local unification"). In some cases this can itself trigger inference (and the cacheing of its results) in other theories, in particular if KM needs to check slot-value constraints on a unified object to make sure the unification is valid in all theories.

Finally, information on any of KM's built-in slots is *always* stored in the main (global) KB, even if that information is entered while being "in" a theory. In other words, this information is necessarily global. In particular, note that the inheritance hierarchy is necessarily global. These built-in slots are (ignoring other internal book-keeping slots): `instance-of`, `instances`, `subclasses`, `superclasses`, `subslots`, `superslots`, `domain`, `range`, `element-type`, `name`, `called`, `uniquely-called`, `members`, `member-of`, `elements`, `combine-values-by-appending`, `inverse`, `inverse2`, `inverse3`, `cardinality`, and `fluent-status`.

24 Morphisms

When building a knowledge-base, the knowledge engineer may find him/herself encoding the same "pattern" of axioms several times. For example, axioms about hydraulic circuits and electrical circuits may both constitute different instantiations of a generic "pattern" about producers and consumers. This pattern itself may constitute an instantiation of a more general pattern about directed graphs. When build a knowledge-base, it is advantageous to make such underlying patterns explicit, and encode them only once so that they can be reused in different contexts. Morphisms provide a mechanism to do this. Creating reusable patterns directly, rather than trying to shoe-horn inheritance to do something similar indirectly, has significant potential for improving reuse in a knowledge-base.

KM's morphism mechanism is implementationally trivial, and simply allows the user to specify *symbol renamings* when importing a set of axioms (i.e. loading a file). Symbol renaming is a syntactic operation (implemented using Lisp's `sublis` function) , and performed during file-load and before the loaded expressions are processed by KM. A symbol renaming table is specified with the keyword `:with-morphism` in a `load-kb` command.

For example, the following file defines the producer-consumer "pattern":

```
;;; ========================================
;;;     File: production-network.km
;;;  Encodes the ‘‘pattern’’ for production
;;; ========================================

(CONNECTS has
  (instance-of (Slot))
  (inverse (CONNECTS)))

(PRODUCER has (superclasses (NODE)))
```

```
(CONSUMER has (superclasses (NODE)))
(INTERMEDIARY has (superclasses (NODE)))

;;; “N reachable from N’ if there’s an unblocked path from N’ to N.”
(every NODE has
  (DIRECTLY-REACHABLE-FROM ((allof (the CONNECTS of Self)
                             where (not (the BLOCKED? of It)))))
  (REACHABLE-FROM ((the DIRECTLY-REACHABLE-FROM of Self)
                   (the REACHABLE-FROM of
                        (the DIRECTLY-REACHABLE-FROM of Self)))))

;;; “A consumer is supplied if its reachable from a producer.”
(every CONSUMER has
  (SUPPLIED? ((if   (oneof (the REACHABLE-FROM of Self)
                    where (It isa PRODUCER))
              then t))))
```

In this file, upper-case class names have been used to denote symbols intended to be “morphed” when this file is imported. (This upper-casing is purely for comprehensibility, and has no special operational significance). The file defines a general pattern of producers and consumers, and the notion of a consumer being supplied. Now, one instantiation of this pattern is in electrical circuits, where an appliance ‘consumer’ is supplied by a power ‘producer’. We can thus build a KB for electrical circuits by importing an morphing this general pattern, rather than writing it from scratch, as follows:

```
;;; ======================================
;;;     File: electricity-network.km
;;;  Instantiate the production "pattern" for electrical circuits
;;; ======================================

(load-kb "production-network.km" :with-morphism
   '((NODE -> Electrical-Device)
     (PRODUCER -> Power-Supply)
     (CONSUMER -> Appliance)
     (INTERMEDIARY -> Switch)
     (SUPPLIED? -> powered?)
     (BLOCKED? -> open?)
     (CONNECTS -> connects)))

(Light has (superclasses (Appliance)))
(Battery has (superclasses (Power-Supply)))
```

(Note that a file can itself contain a `load-kb` command, as above). In this case, the production network file will be read in, but the symbols renamed as listed. We can demonstrate this instantiation by asking whether a light is receiving power or not in a simple electrical circuit:

```
;;; Define the trivial circuit: *Battery1 → *Switch1 → *Light1
KM> (*Battery1 has (instance-of (Battery)))
KM> (*Switch1 has (instance-of (Switch)))
KM> (*Light1 has (instance-of (Light)))
KM> (*Battery1 has (connects (*Switch1)))
KM> (*Switch1 has (connects (*Light1)))
```

```
;;; "Is light1 powered?"
KM> (the powered? of *Light1)
(t)                                         ; Yes!
```

More importantly, we can import the same pattern multiple times, each time under a different morphism. This enables common patterns such as directed graphs, lines, etc. to be reused.

The idea of capturing patterns is a recurring one in AI, and a particularly inspiring paper is Chapman's 1986 memo on cognitive clichés [18]. To our knowledge, KM is one of the few languages which implements this idea, though as described the implementation is trivial and would be easy to reproduce in other KR languages. There are parallels with the work on capturing "design patterns" in object-oriented programming [19]. The mathematical field of Category Theory provides a full, formal foundation for representing, morphing, and composing theory structures together [20], and KM's implementation can be seen as a simple example of this. Our research in this area is described in [21].

25 Procedural Attachment

25.1 Overview

KM allows slots to be filled with a Lisp procedure, which when executed will compute the value(s) of that slot ("procedural attachment"). This can be used, for example, to look up values in a database (rather than translate the entire database into KM structures), or perform algorithmic computation not supported by KM (e.g., matrix arithmetic). Procedural attachment of course requires good knowledge of the Lisp programming language. Caution: procedural attachment should not be used to avoid representing knowledge properly!

An attached procedure must be a Lisp function taking zero arguments, and is placed on the value of a slot. When encountered, KM will execute the function, and it must return a (possibly singleton) *set* of values. If an instance inherits the function as the value of a queried slot, KM will first substitute that instance's name for any occurrence of the keyword `Self` in the function before evaluating it. Syntactically, the function itself is a Lisp lambda expression, which is written in Lisp as `#'(LAMBDA () ...)`, or equivalently `(FUNCTION (LAMBDA () ...))`.

The function itself may require access back into KM, e.g., to look up a slot-value for use in its computation. For these callbacks, use one of the following two Lisp functions:

(km0 '*expr* *[*:fail-mode 'error*]*)
(km-unique0 '*expr* *[*:fail-mode 'error*]*)

Both take a KM expression, and return its evaluation, either as a list of values `(km0 ...)`, or a single value `(km-unique0 ...)`. A fail-mode of `'error` will cause an error to be reported if no values are found, while `'fail` (the default) will just return `NIL`. In addition, `(km-unique0 ...)` will report an error if more than one value is found. Lambda expressions and these functions together allow KM and Lisp to interoperate.

There are two critically important things to remember when using procedural attachment:

1. Ensure that the functions `km0` and `km-unique0`, *not* `km` and `km-unique` (described later), are used for callbacks. `km0` is for within-reasoning calls to KM, and does not reset KM's goal stack, (desirably) allowing KM to detect and recover from looping. `km` is for top-level calls to KM, and does reset KM's goal stack along with other initial computation parameters, and hence should not be used as part of a within-reasoning subgoal.

2. Care must given that upper/lower-case is used correctly in an attached procedure. Unlike the Lisp reader, KM's reader is case-sensitive. Thus, in a KM knowledge-base, any Lisp function names *must* be in upper case. Similarly, within the Lisp code itself, any callbacks to KM must be in the correct case (e.g., lower-case for KM keywords). If the attached procedure is fully specified in the .km file, then KM will automatically read the cases as written. However, if parts of the attached procedure are specified in a .lisp file, then make sure that the case for any callbacks is explicit, as Lisp's reader is not case-sensitive. This can be done by enclosing symbols in vertical bars `||`, or using KM's built-in reader macro `#$` which turns on case-sensitivity in the reader. For example, from the Lisp prompt, the user would have to write:

   ```
   CL-USER> (km0 '(|every| |Car| |has| (|wheels| (4))))
   or
   CL-USER> (km0 '#$(every Car has (wheels (4))))
   ```

   to ensure KM sees the correct case of the expressions.

25.2 Example: Interface to a Database

As an example, suppose a database for material densities exists, and the density of (an object made of) some material can be found using a Lisp call `(get-db` *material* `'density)`. Rather than re-entering the database in KM format, the database can be linked using a function call:

```
(every Homogeneous-object has
  (density (#'(LAMBDA ()
              (LIST (GET-DB (KM-UNIQUE0 '(the material of Self)) 'DENSITY))))))
```

(This example assumes a homogeneous-object is made of a single material). If the density of a particular object is asked about, then `Self` will be replaced by that object's identifier, and the lambda expression evaluated. This expression recursively calls KM to find the object's material ("`(the material of Self)`"), and then gets and returns the density of that material. The final result is returned as a (singleton) value set (list), as required by KM.

25.3 Example: Interface to a Problem-Solving Method

As a second example, imagine that the knowledge engineer has defined a Lisp function `#'PATH-BETWEEN`, which implements a heuristic, graph-traversal problem-solving method (PSM). This function takes as input a start node and a target node, and returns a shortest path between the start and target as a list of traversed nodes. (Its implementation would need to include a callback `km0` to KM, to find out which nodes were connected to any given node). This could be interfaced with KM by defining the concept of a `Graph-Search-Problem`, with slots `start`, `target`, and `shortest-path`, with this function placed on the `shortest-path` slot:

```
(every Graph-Search-Problem has
  (result (#'(LAMBDA ()
              (LET ( (PATH (PATH-BETWEEN
                             (KM-UNIQUE0 '(the start of Self))
                             (KM-UNIQUE0 '(the end of Self)))) )
                (COND (PATH (LIST (CONS ':seq PATH)))))))))
```

Thus:

```
KM> (N1 has (connects (N2)))
KM> (N2 has (connects (N3)))
KM> (N3 has (connects (N4)))
KM> (N2 has (connects (N5)))
KM> (N5 has (connects (N6)))

;;; Invoke the problem-solving method for a specific problem instance.
KM> (the result of (a Graph-Search-Problem with
                      (start (N1))
                      (target (N6))))
((:seq N1 N2 N5 N6))
```

26 Calling KM from External Applications

26.1 KM's API

As well as direct use from the `KM>` prompt, KM can be used as a module within a larger application. The larger application can interface with KM simply by passing KM queries to it and collecting the answers, and thus the minimal API for KM is just two Lisp functions:

`(km '`*expr* *[*`:fail-mode 'error`*]*`)`
`(km-unique '`*expr* *[*`:fail-mode 'error`*]*`)`

These behave in the same way as `km0` and `km-unique0` (Section 25), except they additionally reinitialize the inference engine for a new computation. Both take a KM expression, and return its evaluation, either as a list of values `(km ...)`, or a single value `(km-unique ...)`. A fail-mode of `'error` will cause an error to be reported (i.e., the debugger is activated) if no values are found, while `'fail` (the default) will just return `NIL`. In addition, `(km-unique ...)` will report an error (activate the debugger) if more than one value is found. Again, as Lisp's reader is not case-sensitive, the case of symbols in *expr* must be made explicit in the .lisp files containing the calls to KM, either by surrounding symbols with vertical bars `||`, or using KM's built-in reader macro `#$`.

This minimal API is in theory all that is needed to assert and access any information in the KB. However, some users may prefer a larger list of KM's more specialized, lower-level Lisp functions for accessing and manipulating the KB. These are not provided here, but can be supplied on request.

26.2 Error Handling

By default, if an error occurs during reasoning, KM reports the error and switches on the interactive debugger (tracer), described in Section 6.3. However, when KM is being called from other software, this behavior may be undersirable. To change this behavior, there are two options:

1. Have KM abort and return the error message to the calling software. To do this, set:

   ```
   (setq *abort-on-error-report* t)
   ```

   As a result, top level calls to

   `(km` *expr*`)`
   `(km-unique` *expr*`)`

will return *two* values (use Lisp's `multiple-value-bind` to collect these):

- the answer
- If an error occurred, a string describing the error (NIL otherwise)

Thus the calling software should call KM as follows:

```
(multiple-value-bind
 (answer error-string)
 (km expression)
 ...)
```

`error-string` will be `NIL` if no error occurred, a string otherwise.

2. Have KM ignore the error and continue as if nothing had gone wrong. To do this, set

```
(setq *error-report-silent* t)
```

With this set, no errors will ever be reported to the calling software. Note that there is some risk to using this, as this will not fix the error and other errors may occur as a consequence (which can similarly be ignored). `*error-report-silent*` takes precedence over `*abort-on-error-report*`, if both are set.

26.3 Undoing Operations

In addition, it is sometimes useful for the external application to tell KM to undo (rollback) its computations, if some external condition holds. KM contains an "undo" mechanism for allowing this, described later in Section 28.

26.4 The KM Package

For Lisp users using packages, KM does not have an explicit Lisp package declaration at the start, and so it is loaded into whichever package Lisp is in at the time of the load. During the load of `km.lisp`, KM sets the global variable `*km-package*` to be the current package, i.e., the KM package can be identified by the variable `*km-package*`. Without any package declarations elsewhere, `*km-package*` will be set to the user package.

27 KB Loading and Saving: Advanced Topics

27.1 Incremental Loading and Updating of KBs

27.1.1 Multiple `has` Expressions

Multiple KB files can be loaded using `load-kb`, and the contents of each will be added to memory. If two files refer to the same frame, or more generally if KM encounters two `has` expressions for the same frame, KM will combine the slot-value expressions via unification (`&&`), e.g.:

```
KM> (*Pete has (owns ((a Car))))
KM> (*Pete has (owns ((a Sports-Car))))
KM> (showme *Pete)
(*Pete has
   (owns ((((a Car)) && ((a Sports-Car))))))
```

In this case, asking for `(the owns of *Pete)` will cause the combined expression to be evaluated, and the values will unify (assuming `Sports-Car` is a subclass of `Car`).

27.1.2 `also-has`

This combination behavior can be changed with the forms:

(every *class* also-has *slotsvals*)
(*instance* also-has *slotsvals*)

With these forms, the new *slotsvals* are appended, rather than unified, with the old slotsvals:

```
KM> (*Pete has (owns ((a Car))))
KM> (*Pete also-has (owns ((a Sports-Car))))       ; NB also-has
KM> (showme *Pete)
(*Pete has
   (owns ((a Car) (a Sports-Car))))
```

These forms are useful when the new information is known to be new (rather than a refinement of old information), and are are only intended for use at the `KM>` prompt. It is not recommended to use this form in a KB file (unless the slot values are reified instances) as multiple loads will cause the slots to accumulate multiple copies of the value expressions.

27.1.3 `now-has`

The forms:

(every *class* now-has *slotsvals*)
(*instance* now-has *slotsvals*)

cause the new slot values to overwrite, rather than augment, the old values:

```
KM> (*Pete has (owns ((a Car))))
KM> (*Pete now-has (owns ((a Sports-Car))))       ; NB also-has
KM> (showme *Pete)
(*Pete has
   (owns ((a Sports-Car))))
```

These forms should be used with extreme caution, and only on slots which don't influence any other slots' values (because there is no truth maintenance facility for retracting conclusions based on the deleted facts). It primarily intended for updating simple book-keeping information stored in the KB.

Only slots mentioned in the `now-has` expression will be changed, other slots on the frame will be unaffected. To delete all slot-values, use an empty new slot-value list, e.g., `(*Pete now-has (owns ()))`.

27.2 Controlling the Load Order

When loading a file or multiple files, the order in which KM encounters expressions can occasionally (and undersirably) make a difference to the result. This occurs when a loading expression triggers computation, but the entire KB has not yet been loaded (causing the computation to proceed with incomplete information). Two specific cases of this to be aware of are:

1. KM may not automatically classify an instance as a class (using KM's automatic classification mechanism), because the instance declaration is encountered before the class definition.

2. KM may not remove redundant classes on an `instance-of` or `superclasses` slot, because the full taxonomy has not yet been loaded (and hence the redundancy is not noticed).

The user can avoid these problems by appropriately ordering frames in a KB file, but in cases where the ordering cannot be controlled, the KB may need to be loaded in multiple passes. To support this, `load-kb` can be given a list of *patterns* as an argument, and only expressions matching one of those patterns will be loaded, allowing control over which expressions are loaded when. A pattern is a KM expression containing variables, where a variable is either a symbol starting with a `?` character (matching a single item), or the special symbol `&rest` at the end of a list (matching all remaining items in that list).

If such problems arise, the following is a suitable load order (depending on the KB content):

```
KM> (reset-kb)

;;; 1. load slot declarations and taxonomy KM> (load-kb file :load-patterns '((?x has &rest)))

;;; 2. load classes
KM> (load-kb file :load-patterns '((every &rest)))

;;; 3. load remainder
KM> (load-kb file)
```

Note the last command will include redundant reloading of the previously loaded expressions, but this is not a problem (KM will spot and remove the redundancy at load-time).

27.3 Fast Loading and Saving of KBs

In addition to saving and loading the KB with `save-kb`, `load-kb`, and `reload-kb`, KM also supports fast loading and saving for large KBs. The commands:

```
(fastsave-kb "file.fkm")
(fastload-kb "file.fkm")
```

will save and load the entire KB faster than `save-kb` and `load-kb`, but in a non-human-readable (raw Lisp) form. It is recommended to use the suffixes `".km"` for KM files created by `save-kb` or by hand, and `".fkm"` (fast KM) for files created by `fastsave-kb`.

27.4 Saving and Restoring the KB State to/from Memory

The current state of the KB can also be saved to memory, rather than file, using the command:

```
;;; Store current state of KB to runtime memory
KM> (store-kb)
```

and restored via;

```
;;; Revert back to the last (store-kb) state
KM> (restore-kb)
```

Note that this only stores to RAM, so if you exit Lisp, the stored KB will be lost. Use `save-kb` or `fastsave-kb` for a permanent, file-based save.

For Lisp programmers, the entire KB state can also be captured by the command `(get-kb)`, which returns the large, unreadable Lisp structure representing the current state (namely, a list of Lisp commands for rebuilding the state). `(put-kb` *state*`)` will restore this state (execute these commands), e.g.,:

```
USER: (setq *x* (get-kb))        ;; save state
USER: (put-kb *x*)               ;; restore state
```

Here the state is stored and restored from the global variable `*x*`. The contents of `*x*` are the same as the contents of the "`.fkm`" files created by `fastsave-kb`.

28 Undo

28.1 Overview

KM has a logging mechanism allowing the user to undo commands, including all their side effects, in reverse chronological order. The log records all internal changes to the KB, and hence `(undo)` can incrementally reverse those changes. This is very useful both for debugging, and for an interface for an end-user. For example, if an error occurs during executing a KM command, you can abort from the debugger, type `(undo)`, and then retry your query with tracing.
By default logging is turned off. To turn it on, do:

```
KM> (start-logging)
```

(The opposite command `stop-logging` turns it off again). From this point,

```
KM> (undo)
```

will undo the most recent command and all side-effects. This command can be issued from Lisp directly also. To do this, KM creates a checkpoint in the log at each "`KM>`" prompt, and `(undo)` undoes back to the most recent chackpoint. Multiple `(undo)` are thus possible.

Note that the commands `load-kb`, `reload-kb`, and `reset-kb` will reset the log, so you cannot undo past these commands. Also note that keeping the log is memory-intensive, which is why it is by default turned off.

28.2 For Lisp programmers

For Lisp programmers calling KM from a Lisp program, checkpoints can be manually set using the Lisp function (also KM command):

`(set-checkpoint` *[id]*`)`

where *id* is some name for the checkpoint (any non-nil s-expression). The command

`(undo` *[id]*`)`

will undo back to the checkpoint with name *id*, or back to the most recent checkpoint if no *id* is given.

`(undo)` reverses state changes within KM made between checkpoints, but not any state changes made by the calling Lisp program. In some cases, this is undesirable – there may be state changes made by the calling Lisp program that the programmer would like to be included in the `(undo)`. To deal with this, KM has the function `km-setq`, which is just like Lisp's `setq` except routed through KM's logging mechanism, meaning that `(undo)` will also undo these external setq operations (i.e., put the variables back to their old values) as well as KM's internal operations. Note, however, that with `km-setq` the variable being set *must* be quoted, e.g.:

```
USER: (km-setq '*x* 'cat)           ; Note the variable is quoted with km-setq
```

For example:

```
USER: (setq *x* 'dog)
dog
USER: (start-logging)
USER: (set-checkpoint 'cp1)
USER: (km-setq '*x* 'cat)          ; Note the variable is quoted with km-setq
USER: *x*
cat
USER: (undo)                       ; Undo back to checkpoint cp1
USER: *x*
dog                                ; *x* has reverted back to its old value
```

29 Additional Representational Capabilities

29.1 N-ary Predicates

29.1.1 Representing N-ary Predicates

In most cases, the simplicity of working with just binary predicates is highly advantageous for both knowledge entering and inference. N-ary predicates should normally be handled by the knowledge engineer by reifying them, recasting them as a set of binary predicates. For example, the formula $gives(John, Mary, Book1)$ ("John gives Mary the book.") would be expressed in KM as:

```
KM> (a Giving with
        (agent (*John))
        (recipient (*Mary))
        (patient (*Book1)))
(_Giving23)
```

corresponding to reifying the notion of "a giving" and rewriting the formula as

$$\exists g\ isa(g, Giving) \wedge agent(g, *John) \wedge recipient(g, *Mary) \wedge patient(g, *Book1)$$

However, in occasional circumstances this approach can be counterintuitive or cumbersome. As a result, KM provides support for N-ary predicates as a generalization of its normal slot mechanism. Just as a binary slot contains the values of a predicate's second argument, given the first, so a N-ary slot contains the *value sets* of a predicate's second,...,Nth arguments, given the first. These sets are bundled together in parentheses, and given the keyword `:args` to denote they are a list of arguments (rather than a path).

For example suppose we wish to use the predicate $connects(tank, tank', pipe)$ to denote that $tank$ is connected to $tank'$ via $pipe$, without reifying the concept of "a connection". To do this, we would "bundle" the second and third arguments together as illustrated:

```
;;; "Tank1 is connected to Tank2 via Pipe1"
;;; connects(*Tank1, *Tank2, *Pipe1)
KM> (*Tank1 has
        (connects ((:args *Tank2 *Pipe1))))
```

29.1.2 Accessing N-ary Predicate Arguments

In fact, a `:args` list is a kind of sequence, and so the same forms are used for accessing individual arguments in that list as for sequences. Section 13 described these forms, and they are also illustrated below:

```
;;; Normal query: Will return the 2nd...Nth arguments, given the first.
KM> (the connects of *Tank1)
((:args *Tank2 *Pipe1))

;;; "What is Tank1 connected to?" (return the first argument only)
KM> (the1 of (the connects of *Tank1))
(*Tank2)

;;; (Equivalent shorthand)
KM> (the1 connects of *Tank1)
(*Tank2)

;;; "What is it connected by?" (return 2nd argument only)
KM> (the2 of (the connects of *Tank1))
(*Pipe1)

;;; "Which pipe(s) connect Tank1 to Tank2?"
;;; { p | connects(*Tank1, Tank2, p) }[24]
KM> (the2 of (allof (the connects of *Tank1)
                 where ((the1 of It) = *Tank2)))
(*Pipe1)
```

Finally, just as the `range` is used to specify the range of a predicate's second argument, so `range2`, `range3`, etc. can specify the ranges of the predicate's third, fourth, etc. arguments. If the run-time checking is turned on (with `(checkkbon)`, Section 6.8), then KM will check these ranges are conformed to during inference.

29.1.3 Inverses of N-ary Predicates

For binary slots, we can declare the slot's `inverse` (Section 4), namely the slot corresponding to switching the first and second arguments. Generalizing this, for N-ary predicates we can also define the slot's `inverse2`, corresponding to switching the first and *third* arguments (leaving all others in place), `inverse3` corresponding to switching the first and fourth (leaving all others in place), and `inverse12` corresponding to switching the second and third arguments (leaving all others in place). KM will always install `inverse`s (switching first and second arguments), even if the inverse is not explicitly declared (in which case the inverse slot's name defaults to *slot*`-of`). However, it will not install `inverse2`s and `inverse3`s unless there is an explicit declaration of the name of those inverted slots, indicating that they are of interest to the knowledge engineer. This is to avoid making assertions for all possible permutations of argument orderings for a single ground assertion. For example:

```
;;; Define reaches(from, to, via), and its inverses
;;;      reachable-from(to, from, via), and path-from-X-to-Y(via, to, from)
KM> (reaches has
      (instance-of (Slot))
      (inverse (reachable-from))
      (inverse2 (path-from-X-to-Y)))

;;; "Battery1 reaches light1 via wire1."
;;; reaches(*Battery1, *Light1, *Wire1)
```

---

[24] Or more literally, { $p \mid \exists\, t\ connects(*Tank1, t, p) \wedge t = *Tank2$ }

```
;;; (KM will also automatically install reachable-from(*Light1, *Battery1, *Wire1) and
;;;    path-from-X-to-Y(*Wire1, *Battery1, *Light1))
KM> (*Battery1 has
      (reaches ((:args *Light1 *Wire1))))

;;; (Confirm that the inverse2 was installed)
KM> (showme Wire1)
(*Wire1 has
  (path-from-X-to-Y ((:args *Light1 *Battery1))))
```

Another example using `inverse12` is as follows:

```
KM> (is-between has
       (inverse12 (is-between)))

KM> (*Austin has
       (is-between ((:args *SanAntonio *Dallas))))

KM> (showme *Austin)
(*Austin has
  (is-between ((:args *SanAntonio *Dallas)
               (:args *Dallas *SanAntonio))))
```

29.2 Transitivity

29.2.1 Defining Transitive Relations

Transitive relations (eg. `parts`) should be represented in KM using the standard technique of distinguishing the 'direct' relation (eg. `d-parts`, referring to the *immediate* parts of an object) from the transitive closure of that relation (eg. `parts`). The transitive relation is then defined in terms of the direct relation:

$$\forall x, y \quad d\text{-}parts(x, y) \rightarrow parts(x, y)$$
$$\forall x, y, z \quad d\text{-}parts(x, y) \wedge parts(y, z) \rightarrow parts(x, z)$$

This would be expressed in KM as follows:

```
;;; "An object's parts are its direct parts [1], and the parts of its direct parts [2]."
;;; ∀p isa(p, Physobj) → ( ∀d d-parts(p, d) → parts(p, d) ) ∧
;;;                    ( ∀d, p′ d-parts(p, d) ∧ parts(d, p′) → parts(p, p′) )
(every Physobj has
    (parts ((the d-parts of Self)                      ; [1]
            (the parts of (the d-parts of Self)))))    ; [2]
```

To illustrate this, enter the following KB:

```
(every Physobj has
  (parts ((the d-parts of Self)
          (the parts of (the d-parts of Self)))))

(Car has (superclasses (Physobj)))

(every Car has (d-parts ((a Engine) (a Chassis) (a Body))))
```

```
(Engine has (superclasses (Physobj)))

(every Engine has
    (d-parts ((a Carburetor) (a Battery) (a Combustion-chamber))))

(Body has (superclasses (Physobj)))

(every Body has
  (d-parts ((a Door) (a Door) (a Frame) (a Windshield))))

(Door has (superclasses (Physobj)))

(every Door has
  (d-parts ((a Handle) (a Window) (a Panel))))
```

Thus a query for parts will descend the partonomy, collecting all parts:

```
KM> (the parts of (a Car))
(_Engine415 _Chassis416 _Body417 _Carburetor418 _Battery419
 _Combustion-chamber420 _Door421 _Door422 _Frame423 _Windshield424
 _Handle425 _Window426 _Panel427 _Handle428 _Window429 _Panel430)
```

If just the leaves of the partonomy are required (ie. parts which are not composites of other parts), we can define:

$$(\forall x,y \;\; parts(x,y) \wedge \neg \exists z \;\; d\text{-}parts(y,z)) \rightarrow leaf\text{-}parts(x,y)$$

which in KM can be expressed using a `where` clause (Section 8):

```
;;; "Leaf parts = all the parts [1] which don't have direct parts [2]."
(every Physobj has
  (parts ((the d-parts of Self)
          (the parts of (the d-parts of Self))))
  (leaf-parts ((allof (the parts of Self)                   ; [1]
                 where (not (the d-parts of It))))))         ; [2]
```

Thus we can find all the leaf parts (according to the KB) as follows:

```
KM> (the leaf-parts of (a Car))
(_Chassis433 _Carburetor435 _Battery436 _Combustion-chamber437 _Frame440
_Windshield441 _Handle442 _Window443 _Panel444 _Handle445 _Window446 _Panel447)
```

29.2.2 Built-In Transitive Relations

For convenience, KM has the following built-in slots, which efficiently return the transitive closure of their corresponding relations:

Slot	**Computes transitive closure of:**
`all-classes`	`instance-of`
`all-instances`	`instances`
`all-superclasses`	`superclasses`
`all-subclasses`	`subclasses`
`all-subslots`	`subslots`
`all-superslots`	`superslots`
`all-prototypes`	`prototypes`
`all-supersituations`	`supersituations` (used for situations [8])

29.2.3 “Multi-depth” Paths

Paths of the form (`the` *slot* `of` *expr*) can be thought of as searching one-level deep in a hierarchy made up of *slot* relations. For convenience, KM provides a simple macro expansion to allow searching N levels instead, with a path of the form

(`the` (*slot* `*` *[N]*) `of` *expr*)

(N defaults to 5 if unspecified). This provides an alternative to defining a transitive version of *slot*, although the depth-limit of the search is bounded.

```
;;; "How many parts has my car?"
KM> (the number of (the (d-parts *) of (a Car)))
(16)
```

29.3 Booleans

When evaluating expressions that use logical connectives (`and`, `or`, etc. see Section 7), KM treats any non-NIL value as “true” and the value `NIL` as “false”. For convenience, KM also has the built-in instance `t`, an instance of the class `Boolean`, which can be used as an arbitrary non-NIL symbol to denote “true”.

In exceptional circumstances, using `NIL` to denote false may be undesirable, as it does not distinguish “no result” from “a false result”, and (unlike `t`) cannot be used as a slot-value. For these circumstances, KM also provides the symbol `f`. `f` has no meaning except as a convenient arbitrary symbol to use, and its interpretation as denoting “false” is left to the user. In addition, note that the value `f` does *not* mean “false” to KM’s logical connectives (and, or, not) as it is a non-NIL value, and so care must be taken. An example of using `f` might be to include it in a truth table, whose values are then manipulated by expressions such as `if ((the ...)  = f) then ....`

29.4 Printing

The commands `print`, `format`, and `km-format` are understood by KM, and their “evaluation” produces printed output as a side-effect. Their primary purpose is to support debugging; normally any text output would be generated by some program outside KM, rather than from KM itself. When KM evaluates these forms, it evaluates the value arguments through the normal KM interpreter first, then executes the output command itself with the results, i.e.:

```
KM> (print (a Car))
(_Car6)
(_Car6)

KM> (format t "I have a ~a and a ~a~%" (a Car) (a Book))
I have a (_car11) and a (_book12)
(t)

KM> (km-format t "I have a ~a and a ~a~%" (a Car) (a Book))
I have a (_Car9) and a (_Book10)
(t)
```

`format` is Lisp’s built-in format command. `km-format` is similar, except arguments are written out in a case-sensitive way. `print` returns its evaluated argument, `format` and `km-format` return `t` (so that they can be embedded in a conjuctive `and` expression).

29.5 Quoting Expressions

In some cases, it is desirable to manipulate KM expressions as objects in their own right, rather than have KM evaluate them. To do this, KM expressions can be quoted by preceding them by a single quote (`'`) character. Elements within quoted expressions can be unquoted using the two characters `#,`. The result of evaluating a quoted expression is itself, with the following exception: If there are any unquoted expressions within it, they will be evaluated and the subexpression replaced with the result. This allows KM to represent second order logic expressions, using a quoting/unquoting mechanism similar to Lisp's programmatic quoting/unquoting mechanism.

```
KM> '(1 + 2 + 3)
('(1 + 2 + 3))
```

As a special case, the keyword `Self` in a quoted expression is automatically unquoted, and a "`#,`" prefix is not necessary. It is thus not possible to quote the symbol `Self` itself.

Examples of equivalent quoted expressions are shown below:

```
(1 + (2 + 3) + 4)      =  10
'(1 + (2 + 3) + 4)     =  '(1 + (2 + 3) + 4)
'(1 + #,(2 + 3) + 4)   =  '(1 + 5 + 4)
```

29.6 Aggregates

For slots with multiple values, it is sometimes useful to denote the set ("aggregate") of values as a single representational unit. Aggregates correspond to plurals in English, e.g., we might represent "the wheels of a car" as an aggregate, whose elements are of type (class) `Wheel`. Because an aggregate of wheels is not itself a wheel (say), aggregates need to have special behavior with `(must-be-a` *class*`)` constraints, and to support this KM has a built-in class called `Aggregate`. The sole purpose of the `Aggregate` class is to have KM by-pass the usual `must-be-a` constraint checks. Instead, for `(must-be-a` *class*`)` slot-value constraints applied to aggregates, rather than checking that the aggregate is an instance of *class*, it checks that the `element-type` of the aggregate is not disjoint with *class*, where `element-type` is a built-in slot denoting the class of the aggregate's members. For example:

```
KM> (every Car has
      (parts ((must-be-a Mechanical-Part)
              (a Aggregate with              ; 4 wheels
                (number-of-elements (4))
                (element-type (Wheel)))
              (a Engine))))
KM> (the parts of (a Car))
(_Aggregate2 _Engine3)
```

Note that the `Aggregate` is (desirably) not coerced to be a `Mechanical-Part`. The only built-in slot for `Aggregate` is `element-type`, all other slots on `Aggregate` can be chosen by the user.

29.7 Deleting Frames

The command `(delete` *frame*`)` will delete the frame *frame*, including removing inverse links if *frame* is an instance. This command does *not* undo earlier computations made using *frame* (there is no truth maintenance). Its use is not recommended.

29.8 Forward Chaining on Slots

KM normally operates in a backward-chaining mode, i.e., a slot's value is only evaluated when needed to help conclude another slot's value. However, as an experimental and relatively untested feature, the user can also declare certain slots on a class to be evaluated "opportunistically". For these slots, when an instance of that class is created, KM will *immediately* query these slots, rather than waiting for an explicit query from normal reasoning. To declare these slots for a class, list them on the built-in slot `slots-to-opportunistically-evaluate`, e.g.:

```
(every Property-Value has
   (slots-to-opportunistically-evaluate (value)))
```

Given the above, every time a new instance of `Property-Value` is created, the `value` slot on that instance will be immediately queried by KM.

29.9 Ignoring Inverses

Normally, when KM asserts a ground fact (i.e., computes an instance's slot value) it will also assert the inverse fact (i.e., the value's inverse-slot is filled by instance). To disable this feature, so that inverses are not recorded, the `ignore-inverses` property of that `slot` should be set to `t`:

```
(part-of-speech has
   (instance-of (Slot))
   (ignore-inverses (t)))
```

Given the above, if KM now asserts `(X has (part-of-speech (Y)))`, KM will *not* assert the inverse `(Y has (part-of-speech-of (X)))`.

This feature is for efficiency, and is useful when a slot value is used many times in the KB (and hence the inverse slot would be cluttered with many values). For example, the part of speech `*Noun`) applies to many nouns, and hence the list of values on the inverse slot `(*Noun has (part-of-speech-of (.` would become extremely large if they were all recorded.

KM does not install inverses for the following built-in slots: `aggregation-function`, `called`, `uniquely-called`. `cardinality`, `combine-values-by-appending`, `dont-cache-values`, `ignore-inverses`, `inherit-with-overrides`, `name`, `cloned-from`, `prototype-scope`, `remove-subsumers`, `remove-subsumees`, and (for KM's situation mechanism) `add-list`, `del-list`, `ncs-list`, `pcs-list`. . In addition, to avoid cluttering the KB, KM will not install the inverse link after asserting the following slot-values: `t`, `f`, `1-to-N`, `1-to-1`, `N-to-1`, `N-to-N`, and (for KM's situation mechanism) `*Global`, `*Fluent`, `*Inertial-Fluent`, `*Non-Fluent`.

29.10 Preventing Value Cacheing

Normally, when KM evaluates an expression on a slot, it replaces the expression with the result (i.e., the result is cached). This behavior can be prevented using the slot's slot `dont-cache-values`, e.g.:

```
KM> (text-def has
      (instance-of (Slot))
      (dont-cache-values (t)))
```

This prevents KM cacheing the computed values on this slot, and instead leaves the original expression on the slot. It should only be used in exceptional circumstances, e.g., when a slot's values may change, and nothing else is dependent on those values (as KM does not perform truth maintenance to revise those dependent facts).

29.11 Manipulating Classes as Slot-Values

Classes are themselves instances (of the metaclass `Class`, see Section 29.12), and can be used as slot-values, just like any other instance. For example, "pizza" in "Fred likes pizza" is best viewed as denoting the class `Pizza`, rather than some specific instance of `Pizza`.

For slots which take classes as values, sometimes you may want to retain only the most *specific* classes on the slot, if the semantics of the slot suggest that the more general classes merely give a less precise statement of information which the specific class expresses. These slots can be declared as "remove subsumers" slots, and KM will then remove the subsumers (i.e., any class which subsumes some other class on the slot). `superclasses` is an example of a built-in "remove subumers" slot, e.g., if `Big-Car` has superclasses `Car` and `Vehicle`, then KM will remove the `Vehicle` superclass as it is redundant given the semantics of the slot (and assuming `Vehicle` is a superclass of `Car`).

Conversely sometimes you may want to retain only the most *general* classes on the slot, if the semantics of the slot suggest that the more specific classes merely repeat a narrower part of the information which the general class expresses. These slots can be declared as "remove subsumees" slots, and KM will then remove the subsumees (i.e., any class subsumed by some other class on the slot). `subclasses` is an example of a built-in "remove subumees" slot, e.g., if `Vehicle` has subclasses `Car` and `Big-Car`, then KM will remove the `Big-Car`, as it is redundant given the semantics of the slot.

Remove subsumers/subsumees slots are declared by setting the `remove-subsumers`/`remove-subsumees` property on those slots to `t`. Here is a simple example:

```
;;; Retain only most general class values on this slot
(hates-food-type has (instance-of (Slot)) (remove-subsumees (t)))

(Vegetables has (superclasses (Food)))
(Soggy-Vegetables has (superclasses (Vegetables)))

(every Person has
  (hates-food-type (Soggy-Vegetables)))

(Kid has (superclasses (Person)))

(every Kid has
  (hates-food-type (Vegetables)))
```

Now:

```
;;; What food types does a kid hate?
KM> (the hates-food-type of (a Kid))
(Vegetables)                          ; NB not Soggy-Vegetables too
```

Here, the query answers just `Vegetables`, as the `Soggy-Vegetables` has been removed (it is a subsumee of `Vegetables`). The slot semantics makes this (user-specified) operation valid, as `Soggy-Vegetables` would be redundant given `Vegetables` is already hated.

29.12 Metaclasses

As classes are themselves represented by frames, KM will also support the representation of metaclasses (sets of classes). Note that a metaclass is distinct from a superclass, the former denoting a set of classes, the latter a set of instances. By default, all classes are instances of the metaclass `Class`, however we can also define our own metaclasses, eg.

```
;;; Cars and Vehicles are both instances of vehicle types."
;;; isa(Car, Vehicle-Class)
KM> (Car has (instance-of (Vehicle-Class)))

;;; isa(Van, Vehicle-Class)
KM> (Van has (instance-of (Vehicle-Class)))

KM> (the instances of Vehicle-Class)
(Van Car)
```

This is useful if we want to return an answer which is a class, rather than an instance – the meta-class defines a set of valid answers to search through (each a class). The expression `(an instance of` *class*`)`, below, asserts the existence of an instance[25] of class:

```
;;; "All cars are small."
;;; ∀c isa(c, Car) → size(c, *Small)
KM> (every Car has (size (*Small)))

;;; "Which vehicle types are small?"
;;; { c | isa(c, Vehicle-Class) ∧ ( ∀i isa(i, c) → size(i, *Small) ) }
KM> (allof (the instances of Vehicle-Class)
     where ((the size of (an instance of It)) = *Small))
(Car)
```

29.13 Functions

Some (non-KM) representations make heavy use of functions, e.g., in *height(*Fred)*, *height* is a function which takes a person and returns the person's height. We have described the semantics of KM in terms of binary predicates, e.g., the slot `height` is taken as a binary predicate whose arguments are a person and a height, but one could alternatively interpret KM in terms of functions, e.g., the slot `height` could alternatively be viewed as a function, taking a person as input and returning the person's height. This alternative interpretation of KM's semantics is equally valid, but one we have not used.

In addition, sometimes one might like to use functions within KM expressions. For example, *feet(12)* might denote "12 feet", where *feet* is a function from numbers to distance in feet. KM has a primitive, untested, and purely syntactic way of supporting this by using the prefix `:function`, for example:

```
KM> (*Fred has
      (height ((:function Feet 5))))
```

state that Fred is five feet, where `Feet` can be interpreted as a function applied to `5`. In fact, implementationally `:function` is simply a synonym for `:seq`, and thus this is purely a notational way of creating function-like structures in KM. KM performs no reasoning with these structures, e.g., KM does not realize that `(:function Feet 3)` equals `(:function Yards 1)`. However, it does recognize equality when the structures are identical, e.g.:

```
KM> (*Joe has
      (height ((:function Feet 5))))

KM> ((the height of *Fred) = (the height of *Joe))
(t)
```

---

[25] In this case, a hypothetical instance for reasoning purposes.

This is in contrast to writing:

```
KM> (*Joe has
      (height ((a Length with
                 (magnitude (5))
                 (units (*Feet))))))
```

which makes a different representational decision that each instance of a length is distinct.

As `:function` is a synonym for `:seq`, the user can equivalently use `:seq` or `:pair` to denote a function.

29.14 Introspection

KM provides two (rather rudimentary) forms for retrieving KM expressions themselves from the knowledge base using the KM interpreter, thus allowing "introspection" on its own knowledge. These are:

`(rules-for (the` *slot* `of` *instance*`))`
`(constraints-for (the` *slot* `of` *instance*`))`

The former retrieves all the expressions used to compute an instance's slot values, comprising both local and inherited expressions, and local and inherited constraints. The latter retrieves just the local and inherited constraints which apply to an instance. Results are returned as a set of quoted expressions. For example:

```
KM> (every Car has
       (parts ((a Engine) (must-be-a Engine))))

KM> (rules-for (the parts of (a Car)))
('(a Engine) '(must-be-a Engine))

KM> (constraints-for (the parts of (a Car)))
('(must-be-a Engine))
```

Currently, additional access to expressions and structures within a KM knowledge base needs to be made via Lisp functions in KM's Lisp API (not documented in this manual), rather than through calls to the KM interpreter.

29.15 User-Defined Aggregation Slots

The built-in slots `sum`, `difference`, `product`, `quotient`, `max`, `min`, and `number` (Section 14) are called *aggregation slots*, because they are applied to multiple values and return a single, combined value. In fact the first five are instances of the built-in class `Bag-Aggregation-Slot`, as they take a bag of values as an argument, while `number` is an instance of `Set-Aggregation-Slot`, as it takes a set of values as an argument. The built-in classes `Bag-Aggregation-Slot`, `Set-Aggregation-Slot`, and `Seq-Aggregation-Slot` are subclasses of `Aggregation-Slot`, which is a subclass of `Slot`.

KM also allows the user to define their own set aggregation slots, by specifying the aggregation function on the special slot `aggregation-function`:

```
KM> (max-plus-one has
      (instance-of (Set-Aggregation-Slot))
```

```
      (aggregation-function ('#'(LAMBDA (VALS) (+ (APPLY #'MAX VALS) 1)))))

KM> (the max-plus-one of (:set 1 2 3))
(4)
```

The Lisp function must be quoted, take a single argument (namely the set of values to aggregate), and return a single value (the aggregation). Currently only set aggregation functions can be user-defined (KM does not yet support user-defined bag and sequence aggregation functions).

29.16 User-Defined Infix Operators

KM has several built-in infix operators (`+`, `-`, `isa`, `covers`, etc.), which take two arguments (before and after) and return a result. As a syntactic convenience, KM also allows users to define their own infix operators (which requires knowledge of Lisp). Suppose the user wanted to define `my-older-than`, which took two people as arguments and returned `*Yes` if the first was older than the second, `*No` otherwise, i.e., have KM accept queries like:

```
;;; Is Joe older than Sue?
KM> (*Fred my-older-than *Joe)
(*Yes)
```

To declare `my-older-than` as an infix operator, the global Lisp variable `*user-defined-infix-operators*` must be set to be a list of (for each user-defined infix operator) a pair

(*infix-operator-name lisp-function-implementing-it*)

For example:

```
USER: (setq *user-defined-infix-operators* `((|my-older-than| ,#'my-older-than) ... )
```

declares `my-older-than` as a KM infix operator, whose behavior is defined by the Lisp function `#'my-older-than`. Note the quoting and unquoting with ` and , in the above example so that the function itself, rather than its name, is in the list. Because of this, the `setq` must come *after* the function definitions themselves.

The Lisp implementation of the operator must take exactly two values, which will be fully-evaluated KM expressions (typically two instances, though they could be `(:set ...)` etc constructs also, if the user passes those). KM will call this function with the two objects on the left and right of the infix operator. The function must return a result which is either a single or list of KM values. As for procedural attachment (Section 25), any callbacks to KM from the Lisp code must use the Lisp functions

`(km0 '`*expr* [`:fail-mode 'error`]`)`
`(km-unique0 '`*expr* [`:fail-mode 'error`]`)`

Note *don't* use the functions `(km ...)` or `(km-unique ...)` for callbacks.
In this case, we might implement `#'my-older-than` like this:

```
(defun my-older-than (a b)
  (format t "Calling (my-older-than ~a ~a)...~%" a b)   ; for debugging
  (let ( (age1 (km-unique0 `#$(the age of ,A)))         ; in this toy, returns a number
         (age2 (km-unique0 `#$(the age of ,B))) )
    (cond ((and (numberp age1) (numberp age2))
           (cond ((> age1 age2) '#$*Yes)
                 ((<= age1 age2) '#$*No)))
          (t '#$*DontKnow))))
```

with the resulting behavior:

```
KM> (*Fred has (age (20)))
KM> (*Joe has (age (19)))
KM> (*Fred my-older-than *Joe)         ; Is Fred older than Joe?
Calling (my-older-than *Fred *Joe)...
(*Yes)

KM> (*Joe my-older-than *Fred)         ; Is Joe older than Fred?
Calling (my-older-than *Joe *Fred)...
(*No)

KM> (*Joe my-older-than *Sue)          ; Is Joe older than Sue?
Calling (my-older-than *Joe *Sue)...
(*DontKnow)
```

29.17 Identifying Anonymous Instances

The test `(anonymous-instancep` *instance*`)` returns `t` if *instance* is anonymous (starts with a _), `NIL` otherwise.

29.18 KM Versioning Control

The command `(version)` displays the KM version you are running. To ensure that your KB is using a sufficiently up-to-date version of KM, you can add the following to your KB (or Lisp code):

```
(requires-km-version "1.4.5.9")
```

When encountered, KM will check its current version is at least as recent as the one specified. If it is not, it will abort with a friendly message requesting you to upgrade your KM version.

30 KM: Known Limitations

30.1 Sources of Incompleteness and Incorrectness

There is a well-known trade-off in AI between having a language which is *expressive*, and one which has a *tractable*, logically complete inference procedure associated with it. KM tries to maintain a balance in this respect by using a language which is expressive, and an inference procedure which is 'reasonably complete'; that is, KM can infer many, but not all, logical consequences of a KB. The degree to which incompleteness is a practical problem depends on how the KB is built, and whether expressions can be specified such that logical incompleteness has minimal effect on question-answering ability. As KM makes the closed-world assumption and treats negation as failure, sources of incompleteness can in principle also cause incorrectness.

30.1.1 "Hidden Inverses"

In the absence of other information, KM will not realize that the inverse relation $R^{-1}(I_2, I_1)$ holds between two instances until the 'forward' relation $R(I_1, I_2)$ has been computed. An example of this is:

```
;;; connected-to is a reflexive relation (inverse is the same)
```

```
KM> (connected-to has
      (inverse (connected-to)))

;;; Every car has an engine, and a fuel-tank connected to that engine.
KM> (every Car has
      (parts ((a Engine)
              (a Fuel-Tank with
                 (connected-to ((the Engine parts of Self)))))))

KM> (a Car)
(_Car1)

KM> (the parts of _Car1)
(_Engine2 _Fuel-Tank3)

;;; What is _Engine2 connected to?
;;; Answer should be _Fuel-Tank3! KM doesn't realize this fact as the
;;; expression on Fuel-Tank has not yet been evaluated.
KM> (the connected-to of _Engine2)
NIL

;;; What is the fuel tank connected to? (Triggers evaluation of the expression)
KM> (the connected-to of _Fuel-Tank3)
(_Engine2)

;;; Now that KM knows the Fuel-Tank is connected to the Engine, it also
;;; knows the inverse (that the Engine is connected to the Fuel-Tank)
KM> (the connected-to of _Engine2)
(_Fuel-Tank3)
```

In this example, KM does not realize the engine is connected to the fuel-tank until it computes that the fuel-tank is connected to the engine, i.e., it does not "realize" that a computation elsewhere in the KB might indirectly help it conclude what the engine is connected to.

In our experience, problems from this kind of incompleteness have been rare. In cases where it may be a problem, it can be avoided by stating axioms in both forward and backward directions of the predicate, eg:

```
KM> (every Car has
      (parts ((a Engine with
                 (connected-to ((the Fuel-Tank parts of Self))))
              (a Fuel-Tank with
                 (connected-to ((the Engine parts of Self)))))))
```

Alternatively, controlled forward chaining or use of prototypes can remove or reduce this source of incompleteness.

30.1.2 Non-Local Effects do not Trigger Classification

Every time there is a change to an instance (ie. one of its slot-values is computed or asserted), KM rechecks that the instance is still correctly classified. However, if there is a change *elsewhere* in the KB, for example a slot-value of one of its slot-values is computed/asserted (ie. one step removed from a change on its immediate slots), KM will not recheck that instance's classification.

In some cases, it is possible that these "non-local" effects imply that instance's (most specific) class should change, but KM will not realize this. For example, consider the definition below:

```
;;; "Fast cars are cars with big engines."
;;; ∀f isa(f,Fast-Car) ↔ ∃e isa(f,Car) ∧ isa(e,Engine) ∧ engine(f,e) ∧ size(e,*Big)
KM> (every Fast-Car has-definition
        (instance-of (Car))
        (engine ((a Engine with (size (*Big))))))
Noting definition for Fast-Car...

;;; "My car has an engine called *My-Engine."
KM> (*My-Car has (instance-of (Car)) (engine (*My-Engine)))
```

Suppose we now assert that `*My-Engine` (which is the engine of `My-Car`) is big:

```
;;; "*My-Engine is big."
KM> (*My-Engine has (instance-of (Engine)) (size (*Big)))
```

According to the definition of `Fast-Car`, `*My-Car` is now a `Fast-Car`. But KM will not reclassify `*My-Car`, as `*My-Car` itself wasn't directly touched by this last assertion (we only touched `*My-Engine`). As a result, KM is unaware that `Fast-Car` properties now hold for `*My-Car`. Only when a subsequent direct change is made to `*My-Car` will KM check it is properly classified, and here find its most specific class (`Car`) is overly general (and hence specialize it), eg.:

```
;;; "*My-Car is red."
;;; As this touches the *My-Car instance, KM rechecks its classification and
;;; only now realizes *My-Car's class can be refined.
KM> (*My-Car has (color (*Red)))
(*My-Car satisfies definition of Fast-Car:
 Changing *My-Car's classes from (Car) to (Fast-Car))
```

30.1.3 Heuristic Unification

As described in Section 10, KM's set unification is (deliberately) heuristic rather than deductive. As a result, KM's heuristics may incorrectly assume coreference. Section 10.3.2 discussed this in more detail, and how heuristic unification can be more tightly controlled if necessary.

30.1.4 Mutually Inconsistent Constraints

KM currently will not spot mutually inconsistent constraints during unification (e.g., `must-be-a` and `mustnt-be-a` constraints), it will only spot when a slot's constraint conflicts with a slot's instance (and hence block the unification).

30.2 Other Implementational Limitations

30.2.1 No Truth Maintenance

KM's reasoning is essentially monotonic, and KM does not have a truth maintenance capability built into it (although "weak" techniques are available such as full rollbacks of the KB state via `(undo)`, or explicit KB state editing via `now-has` assertions). Thus for a KB where the user is incrementally supplying more information, KM is not able to recognize where old conclusions are no longer valid (e.g., a conclusion based on negation as failure or heuristic unification), and that it should be retracted.

30.2.2 Unifying Sequences and bags

The following set unification of two sequences generates an error, rather than the set of the two sequences:

```
KM> (((:seq _X 1)) && ((:seq 2 _X)))
ERROR! Yikes! I partly unified two sequences...
```

In this case, KM recognizes too late that a multiply occurring variable can only be consistently bound locally, but not globally.

Bags are currently unified as if they were sequences, a limitation of the current implementation.

Acknowledgements

We are greatly indebted to all the individuals who have helped in the evolution of KM, including Ken Barker, Don Batory, Charles Callaway, Vinay Chaudhri, Eric Eilerts, Adam Farquhar, Jihie Kim, Ben Kuipers, Jo Lee, James Lester, Vladimir Lifschitz, Rich Mallory, Ken Murray, Norman McCain, Anand Padmanaban, Jeff Provost, Jeff Rickel, Art Souther, Rich Sutton, and John Thompson.

References

[1] D. Bobrow and T. Winograd. An overview of KRL, a knowledge representation language. In R. Brachman and H. Levesque, editors, *Readings in Knowledge Representation*, pages 264–285. Kaufmann, CA, 1985. (originally in Cognitive Science 1 (1), 1977, 3–46).

[2] R. MacGregor and R. Bates. The LOOM knowledge representation language. Tech Report ISI-RS-87-188, ISI, CA, 1987.

[3] R. J. Brachman, D. L. McGuinness, P. F. Patel-Schneider, L. A. Resnick, and A. Borgida. Living with CLASSIC: When and how to use a KL-ONE like language. In J. Sowa, editor, *Principles of Semantic Networks*. Kaufmann, CA, 1991.

[4] Tom M. Mitchell, John Allen, Prasad Chalasani, John Cheng, Oren Etzioni, Marc N. Ringuette, and Jeffrey C. Schlimmer. Theo: A framework for self-improving systems. In K. Vanlehn, editor, *Architectures for Intelligence*. Erlbaum, 1991. (http://www.erlbaum.com/1652.htm).

[5] B. W. Porter, J. Lester, K. Murray, K. Pittman, A. Souther, L. Acker, and T. Jones. AI research in the context of a multifunctional knowledge base: The botany knowledge base project. Tech Report AI-88-88, Dept CS, Univ Texas at Austin, Sept 1988.

[6] Bruce Porter. The knowledge representation group, UT Austin. http://www.cs.utexas.edu/users/mfkb/, 1996.

[7] DARPA. The rapid knowledge formation project (web site). http://reliant.teknowledge.com/RKF/, 2000.

[8] Peter Clark and Bruce Porter. KM – situations, simulations, and possible worlds. Technical report, AI Lab, Univ Texas at Austin, 1999. (http://www.cs.utexas.edu/users/mfkb/km.html).

[9] Raymond M. Smullyan. *First-Order Logic*. Dover, NY, 1995.

[10] Peter Clark and Bruce Porter. KM – the knowledge machine: Reference manual. Technical report, AI Lab, Univ Texas at Austin, 1999. (http://www.cs.utexas.edu/users/mfkb/km.html).

[11] Timothy Budd. *An Introduction to Object-Oriented Programming*. Addison-Wesley, MA, 1991.

[12] Earnest Davis. *Representations of Commonsense Knowledge*. Kaufmann, CA, 1990.

[13] J. M. Crawford and B. J. Kuipers. Algernon – a tractable system for knowledge-representation. *SIGART Bulletin*, 2(3):35–44, June 1991.

[14] Peter Clark and Bruce Porter. Building concept representations from reusable components. In *AAAI-97*, pages 369–376, CA, 1997. AAAI.

[15] P. Clark, J. Thompson, K. Barker, B. Porter, V. Chaudhri, A. Rodriguez, J. Thomere, S. Mishra, Y. Gil, P. Hayes, and T. Reichherzer. Knowledge entry as the graphical assembly of components. In *Proc 1st Int Conf on Knowledge Capture (K-Cap'01)*, pages 22–29. ACM, 2001.

[16] John McCarthy and Sasa Buvac. Formalizing context. In *AAAI Fall Symposium on Context in Knowledge Representation*, pages 99–135. AAAI, 1997.

[17] Paul Blair, R. V. Guha, and Wanda Pratt. Microtheories: An ontological engineer's guide. Tech Rept CYC-050-92, MCC, Austin, TX, Mar 1992.

[18] David Chapman. Cognitive cliches. AI Working Paper 286, MIT, MA, Apr 1986.

[19] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns.* Addison-Wesley, 1995.

[20] B. Pierce. *Basic Category Theory for Computer Scientists.* MIT Press, 1991.

[21] Peter Clark, John Thompson, and Bruce Porter. Knowledge patterns. In A. Cohn, F. Giunchiglia, and B. Selman, editors, *Proc 7th Int Conf on Knowledge Representation and Reasoning (KR'2000)*, pages 591–600, CA, 2000. Kaufmann.

Index