# KM – The Knowledge Machine 1.4.0: Reference Manual

(Revision 1, for KM 1.4.0 and later.
See release notes for recent updates)

Peter Clark
Mathematics and Computing Technology
The Boeing Company
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

KM is the language used by the Knowledge Representation Group at University of Texas at Austin. It is a frame-based representation language in the spirit of KRL [1], and has some similarities with KL-ONE representation languages such as LOOM [2] and CLASSIC [3]. This document is a Reference Manual for the language, and is meant to accompany the Users Manual [4] and Situations Manual [5] which provide more details on how KM can be used. All these manuals, the example KBs for them, and the KM implementation itself, are available at http://www.cs.utexas.edu/users/mfkb/km.html.

This Reference Manual gives a very brief summary of all the different forms which KM accepts. The User Manual, Situations Manual, and Release Notes give full details of their meanings.

# 2 A BNF for KM

**Queries:**

*expr* =

| | |
|---|---|
| *instance* \| | ; an atomic instance |
| *class* \| | ; a class |
| `(:set` *expr**`)` \| | ; a set of expressions |
| `(:seq` *expr**`)` \| | ; a sequence of expressions |
| `(:args` *expr**`)` \| | ; a multi-argument structure |

**Assertions: Anonymous Instance Creation:**

| | |
|---|---|
| `(a` *class* *[*`with` *slotsvals]*`)` \| | ; Create instance of *class* (plus give it some slot-values) |
| `(an instance of` *expr*`)` \| | ; Create instance of class *expr* |

**Assertions: Named Instance and Class Creation:**

| | |
|---|---|
| `(`*expr* `has` *slotsvals*`)` \| | ; Declare slot-values for the instance/class *expr* |
| `(every` *expr* `has` *slotsvals*`)` \| | ; Declare slot-values for members of the class *expr* |

**Queries: Path Following:**

| | |
|---|---|
| `(the` *slot* `of` *expr*`)` \| | ; Find values of *slot* on instance *expr* |
| `(the` *class slot* `of` *expr*`)` \| | ; Same, but select only values in *class* |
| `(`*expr* $slot_1$ $class_1$ ... $slot_n$ *[*$class_n$*]*`)` \| | ; Same (alternative 'linear' syntax) |

**Conditional and Boolean Expressions:**

| | |
|---|---|
| `(if` *expr1* `then` *expr2 [*`else` *expr3]*`)` \| | ; if *expr1* evals to non-nil, eval *expr2* else eval *expr3* |
| `(`*expr* `and` *expr*`)` \| | ; Conjunction |
| `(`*expr* `or` *expr*`)` \| | ; Disjunction |
| `(not` *expr*`)` \| | ; Negation (Using negation as failure) |
| `(numberp` *expr*`)` \| | ; Test if *expr* is a number |
| `(`*expr* `=` *expr*`)` \| | ; Test equality |
| `(`*expr* `/=` *expr*`)` \| | ; Test inequality |
| `(`*expr* $>$ *expr*`)` \| | ; arithmetic comparison |
| `(`*expr* $<$ *expr*`)` \| | |
| `(`*expr* $>=$ *expr*`)` \| | |
| `(`*expr* $<=$ *expr*`)` \| | |

(*expr1* `includes` *expr2*) | ; Set *expr1* includes instance *expr2*
(*expr* `is-superset-of` *expr*) | ; Set *expr1* includes set *expr2*
(*expr* `isa` *class*) | ; *expr* is an instance of *class*
(`has-value` *expr*) | ; *expr* evaluates to a non-nil value (equivalent to *expr*)

**'Forall' Expressions:**

(`allof` *expr1* `where` *expr0*) | ; Find all *expr1* members passing test *expr0*
(`forall` *expr1* [`where` *expr0*] *expr2*) | ; Eval *expr2* for all *expr1* [passing test *expr0*]
(`oneof` *expr1* `where` *expr0*) | ; First *expr1* member passing test *expr0*
(`theoneof` *expr1* `where` *expr0*) | ; The *expr1* member passing test *expr0*
(`allof` *expr1* [ `where` *expr2*] `must` *expr2*) | ; Check all *expr1* members passing
; test *expr2* also pass *expr0*
(`allof2` *expr1* `where` *expr0*) | ; Same, except keyword '`It2`' rather than
(`forall2` *expr1* [`where` *expr0*] *expr2*) | ; '`It`' denotes the referent
(`oneof2` *expr1* `where` *expr0*) |
(`theoneof2` *expr1* `where` *expr0*) |
(`allof2` *expr1* [ `where` *expr2*] `must` *expr2*) |

**Arithmetic Computations:**

(*expr op expr [op expr]**) | ; Arithmetic, where *op* is one of `+`,`-`,`*`,`/`,`^`
(`the sum of` *expr*) | ; Add, alternative form (*expr* evals to a set of numbers)
(`the difference of` *expr*) | ; Subtract: (((n1 - n2) - n3) - ... - nN)
(`the product of` *expr*) | ; Multiply
(`the quotient of` *expr*) | ; Divide: (((n1 / n2) / n3) / ... / nN)
(`the max of` *expr*) | ; The maximum of the value(s) *expr*
(`the min of` *expr*) | ; Minimum
(`the average of` *expr*) | ; Average
(`the number of` *expr*) | ; Number of values *expr* returns
(`the abs of` *expr*) | ; Remove negative sign from *expr* eg. -1 → 1
(`the floor of` *expr*) | ; Remove decimals from *expr*, eg. 1.03 → 1
(`the log of` *expr*) | ; Logarithm (base e)
(`the exp of` *expr*) | ; Exponent
(`the sqrt of` *expr*) | ; Square root

**The Object Stack ("Contexts"):**

(`new-context`) | ; Clear the object stack
(`show-context`) | ; List all the instances on the object stack
(`the` *class* [`with` *slotsvals*]) | ; Find the instance (on the object stack) of *class* with *slotsvals*
(`every` *class* [`with` *slotsvals*]) | ; Find all instances (on the object stack) of *class* with *slotsvals*
(`the+` *class* [`with` *slotsvals*]) | ; Find-or-create the instance of *class* with given *slotsvals*
(`a+` *class* [`with` *slotsvals*]) | ; Same (Synonym for `the+`)
(`thelast` *class*) | ; Find the most recent instance of *class* on the object stack

**Unification:**

(*expr* `&` *expr* [`&` *expr*]*) | ; unify instances
(*expr* `==` *expr* [`==` *expr*]*) | ; unify instances (synonym for `&`)
(*expr* `&!` *expr* [`&!` *expr*]*) | ; unify instances eagerly

(*expr* `&?` *expr*) | ; test if instances will unify
(`the unification of` *expr*) | ; Unify (using `&`) all elements in *expr*

((*expr**) `&&` (*expr**) *[*`&&` (*expr**)*]**) | ; unify sets
((*expr**) `===` (*expr**) *[*`===` (*expr**)*]**) | ; unify sets (synonym for `&&`)
((*expr**) `&&!` (*expr**) *[*`&&!` (*expr**)*]**) | ; unify sets eagerly
(`the set-unification of` *expr*) | ; Unify (using `&&`) all values which *expr* returns

**Constraints on Slot-Values:**

(`must-be-a` *class* *[*`with` *slotsvals]*) | ; All values must be subsumed by this description
(`mustnt-be-a` *class* *[*`with` *slotsvals]*) | ; No values can be subsumed by this description
(`<>` *expr*) | ; No values can be equal to *expr*
(`constraint` *expr*) | ; *expr* must be true for every value (denoted by `TheValue` in *expr*)

**Constraints on the Set of Slot-Values:**

(`at-least` *integer class*) | ; Must contain at least *integer* instances of *class*
(`at-most` *integer class*) | ; Must contain at most *integer* instances of *class*
(`exactly` *integer class*) | ; Must contain exactly *integer* instances of *class*
(`set-constraint` *expr*) | ; *expr* must be true for the value set (denoted by `TheValues` in *expr*)

**Classification:**

(`every` *class* `has-definition` *slotsvals*) | ; Condition for membership
(*instance* `has-definition` *slotsvals*) | ; Condition for equivalence

**Sequences:**

(`the1 of` *expr*) | ; First element of `:args`/`:seq` (sequence) structure
(`the2 of` *expr*) | ; Second element of `:args`/`:seq` (sequence) structure
(`the3 of` *expr*) | ; Third element of `:args`/`:seq` (sequence) structure
(`theN` *N* `of` *expr*) | ; *N*th element of `:args`/`:seq` (sequence) structure
(`the1` *slot* `of` *expr*) | ; First element of the `:args` structure on *expr*'s *slot*
(`the2` *slot* `of` *expr*) | ; Second element of the `:args` structure on *expr*'s *slot*
(`the3` *slot* `of` *expr*) | ; Third element of the `:args` structure on *expr*'s *slot*

**Text Generation:**

(`the name of` *expr*) | ; special slot: generate a name for *expr*
(`make-phase` *expr*) | ; Convert sequence (`:seq`) of strings + instances to phrase.
(`make-sentence` *expr*) | ; Convert sequence to sentence (capitalize and add a '.')
(`andify` *expr*) | ; `(a b c)` becomes "`a, b, and c`"
(`pluralize` *expr*) | ; "car" becomes "cars"
(`print` *expr*) | ; Print the result of evaluating *expr*
(`format t` *string expr**) | ; Print *string* using Lisp's *format* command, substituting
; (evaluated) *expr** for "~a"s in *string*.
(`format nil` *string expr**) | ; Same, but return rather than print the string.
(`km-format` *[*`t`|`nil`*]* *string expr**) | ; Same, but better formatting control for "~a".

**Queries: Quoted Expressions and Subsumption:**

' *expr* | ; a quoted expression (does not get evaluated)

`#,`*expr* | ; (Within a quoted expression) unquote (i.e. evaluate) *expr*
`(evaluate` *expr*`)` | ; evaluate the quoted expression(s) which *expr* evaluates to
`(`*expr1* `subsumes` *expr2*`)` | ; Class description *expr1* subsumes class description *expr2*
`(`*expr1* `covers` *expr2*`)` | ; Instance description *expr2* is in class description *expr1*
`(`*expr1* `is` *expr2*`)` | ; (Same as subsumes, but with instance descriptions)

**Prototypes:**

`(a-prototype` *class [*`with` *slotsvals]*`)` | ; Create a prototype of *class* and enter prototype mode
`(end-prototype)` | ; Exit prototype mode
`(clone` *expr*`)` ; Clone the prototype instance *expr*

**Propositions:**

`(:triple` *expr expr expr*`)` | ; A frame-slot-value triple
`(the frame of` *expr*`)` | ; The frame in the triple *expr*
`(the slot of` *expr*`)` | ; The slot in the triple *expr*
`(the value of` *expr*`)` | ; The value in the triple *expr*
`(assert` *expr*`)` | ; Assert triple *expr* in the KB
`(is-true` *expr*`)` | ; The triple *expr* holds in the KB
`(all-true` *expr*`)` | ; The triple(s) *expr* all hold in the KB
`(some-true` *expr*`)` | ; At least one of the triple(s) *expr* hold in the KB

**Situations:**

`(new-situation)` | ; Create and enter a new situation
`(next-situation)` | ; Create and enter the temporally next situation
`(global-situation)` | ; Return to the global situation
`(curr-situation)` | ; Evaluates to the current situation
`(in-situation` *expr*`)` | ; Enter situation *expr*
`(in-situation` *expr1 expr2*`)` | ; Evaluate *expr2* in situation *expr1*
`(in-every-situation` *situation-class expr*`)` | ; *expr* holds in all situations of *situation-class*
`(do` *expr*`)` | ; Create next situation by doing action *expr*
`(do-and-next` *expr*`)` | ; Create and enter next situation by doing action *expr*
`(do-script` *expr*`)` | ; ≡ `(forall (the actions of` *expr*`) (do-and-next It))` (Obsolete)
`(default-fluent-status` *[fluent-status]*`)` | ; View/set default fluent status for slots
`(some` *class [*`with` *slotsvals]*`)` | ; Create a fluent instance (experimental)

**Escape to Lisp:**

*lisp-function* | ; execute *lisp-function*

**Other Commands: Loading and Saving a KB**

`(reset-kb)` | ; Delete the current KB from memory
`(load-kb` *filename [*`:verbose t`*] [*`:with-morphism` *morphism]*`)` | ; Evaluate all exprs in a file
`(reload-kb` *filename [*`:verbose t`*] [*`:with-morphism` *morphism]*`)` | ; Same, but `(reset-kb)` first
`(save-kb` *filename*`)` | ; Write current KB to a file
`(write-kb)` | ; Write current KB to standard output

**Other Commands: General**

`(ignore-result` *expr*`)` | ; Evaluate *expr* then return `(t)`, regardless of the result.

`(delete` *expr*`)` | ; Delete the frame *expr* (NB but not dependent facts!)
`(reverse` *expr*`)` | ; Reverse the sequence *expr* (a `(:seq i1 ... in)` expr).
`(evaluate-paths)` | ; Evaluate all unexpanded paths cached on instances in current context
`(graph` *expr [depth]*`)` | ; Print a graph of instance *expr* to depth *depth* (an integer)
`(showme` *expr*`)` | ; Display slot-values of *expr*
`(showme-all` *expr*`)` | ; Display all slot-values of *expr* (including nils)
`(evaluate-all` *expr*`)` | ; Compute and display all slot-values of *expr*
`(showme-here` *expr*`)` | ; Display slot-values of *expr* in the current situation only
`(taxonomy)` | ; Print the isa hierarchy
`(show-bindings)` | ; Display all variable bindings
`(trace)` | ; Turn on tracing
`(untrace)` | ; Turn off tracing
`(checkkbon)` | ; Turn on run-time checking of the KB
`(checkkboff)` | ; Turn off run-time checking of the KB
`(install-all-subclasses)` | ; Re-compute subclass links from superclass links
`(scan-kb)` | ; Cursory check of KB for undefined symbols
`(disable-classification)` | ; Switch off KM's classification mechanism
`(enable-classification)` | ; Switch it on again
`(fail-noisily)` | ; Treat an answer NIL as an error
`(fail-quietly)` | ; Treat an answer NIL as okay (default)
`(nocomments)` | ; Suppress KM's printing of comments during inference
`(comments)` | ; Switch on KM's printing of comments during inference
`(setq` *var val*`)` ; (Lisp) Set Lisp variable *var* to *val* (*var* & *val* are *symbol*s)

**Sub-expressions:**
*slotsvals* = *slotvals**
*slotvals* = `(`*slot* `(`*expr**`))` ; eg. `(pets ((a Cat) (a Dog with (age (33)))))`
*slot* = *symbol* | `(`*symbol* `*` *[n]*`)` ; (Latter is a multidepth path)
*class* = *symbol*
*instance* =
`_`*namenumber* | ; anonymous instance eg. `_Car33`
`_Proto`*namenumber* | ; prototype instance eg. `_ProtoCar33`
`_Some`*namenumber* | ; fluent instance eg. `_SomeCar33` (experimental)
*string* |
*number* |
*symbol* ; named instance (recommended to prefix with a `*`, eg. `*Pete`)
*filename* = *string*
*lisp-function* = `#'`*sexpr* | `(function` *sexpr*`)` ; *sexpr* is a Lisp S-expression
*symbol* = a Lisp symbol

**Lisp Commands** (Access to KM from the Lisp Prompt):
`(km)` | ; start KM interpreter
`(km '#$`*expr* [ `:fail-mode 'fail` ]`)` ; Evaluate *expr* from Lisp prompt (`#$` for case-sensitivity)

**Keywords** (denoting the instance under consideration):
`Self` ; in `(every` *class*`...)` expressions.
`It` ; in `allof/oneof/theoneof/forall/forone` expressions.

```
It2                    ; in allof2/oneof2/theoneof2/forall2/forone2 expressions.
TheValue               ; in (constraint ...) expressions.
TheValues              ; in (set-constraint ...) expressions. (Denotes the set of slot-values)
TheSituation           ; in (in-every-situation ...) expressions. (Denotes the situation)
```

# 3 Built-in Classes and Instances

KM's built-in taxonomy is as follows, showing all but KM's built-in slot instances (these are listed in the next Section). `I` denotes instances, rest are classes. Indentation shows the `subclasses`/`instances` relationships.

```
Thing
   Boolean
I    t
I    f
   Cardinality
I    1-to-1
I    1-to-N
I    N-to-1
I    N-to-N
   Class
   Fluent-Status
I    *Fluent
I    *Inertial-Fluent
I    *Non-Fluent
   Number
     Integer
   Partition
   Situation
I    *Global
   Slot
     Aggregation-Slot
   String
```

# 4 Built-in Slots

| Name | (Applied to) Purpose |
|---|---|
| `abs` | (Number) remove any negative sign |
| `add-list` | (Action) triples which an action makes true |
| `aggregation-function` | (Aggregation Slot) function to use to aggregate values |
| `all-classes` | (Instance) All the classes of an instance |
| `all-instances` | (Class) All the instances of a class (both direct and indirect) |
| `all-prototypes` | (Class) All the prototypes of a class |
| `all-subclasses` | (Class) All the subclasses of a class |
| `all-subslots` | (Slot) All the subslots of a slot |
| `all-superclasses` | (Class) All the superclasses of a class |
| `all-supersituations` | (Situation) All the supersituations of a situation |
| `assertions` | (Situation, book-keeping) Assertions to make in a new situation |
| `average` | (Numbers) Average of a set of numbers |

| Name | (Applied to) Purpose |
|---|---|
| `cardinality` | (Slot) Cardinality restrictions on a slot |
| `classes` | (Instance) Immediate classes of an instance (same as `instance-of`) |
| `cloned-from` | (Instance, book-keeping) source prototype(s) for an instance |
| `definition` | (Prototype, book-keeping) definitional properties of a prototype |
| `del-list` | (Action) triples which an action makes false |
| `difference` | (Numbers) Difference of a set of numbers (((n1-n2)-n3)-...-nN) |
| `domain` | (Slot) class restriction on a slot's first argument |
| `domain-of` | (Class) inverse of `domain` |
| `elements` | (Sequence) Return the elements in a sequence as a set |
| `exp` | (Number) exponent |
| `fifth` | (Set) Fifth element in a set of values† |
| `first` | (Set) First element in a set of values† |
| `floor` | (Number) Remove decimals, e.g., 1.03 → 1 |
| `fluent-status` | (Slot) Declare if slot is a `*Fluent`, `*Non-Fluent` or `*Inertial-Fluent` |
| `fluent-status-of` | (Fluent-Status) Inverse of `fluent-status` |
| `fourth` | (Set) Fourth element in a set of values† |
| `instance-of` | (Instance) The immediate classes of an instance |
| `instances` | (Class) The immediate instances of a class |
| `inverse` | (Slot) Name of the slot's inverse slot |
| `inverse2` | (Slot) Name of an N-ary slot's "second inverse" |
| `inverse3` | (Slot) Name of an N-ary slot's "third inverse" |
| `last` | (Set) Last element in a set of values† |
| `log` | (Number) Logarithm of a number |
| `max` | (Numbers) Maximum of a set of numbers |
| `members` | (Partition) Classes in the partition |
| `min` | (Numbers) Minimum of a set of numbers |
| `name` | (Instance) Pretty name (text fragments) of an instance |
| `ncs-list` | (Action) triples false before an action happens |
| `next-situation` | (Situation) The temporally next situation |
| `number` | (Set) The number of values in a set |
| `pcs-list` | (Action) triples true before an action happens |
| `prev-situation` | (Situation) The temporally previous situation |
| `product` | (Numbers) Product of a set of numbers |
| `protopart-of` | (Prototype, book-keeping) Inverse of `protoparts` |
| `protoparts` | (Prototype, book-keeping) Instances which are part of the prototype |
| `prototype-of` | (Prototype, book-keeping) Class which the prototype is of |
| `prototypes` | (Class) Prototypes of a class |
| `quotient` | (Numbers) Quotient of a set of numbers (((n1/n2)/n3)/.../nN) |
| `range` | (Slot) class restriction on a slot's second argument |
| `range-of` | (Class) inverse of `range` |
| `second` | (Set) Second element in a set of values† |
| `set-unification` | (Set) Unify (using `&&`) all the members of a set |
| `situation-specific` | (Slot) If `t`, do not compute slot values in global situation |
| `sqrt` | (Number) The square root of a number |
| `subclasses` | (Class) Immediate subclasses of a class |

| Name | (Applied to) Purpose |
|---|---|
| `subsituations` | (Situation) Immediate subsituations of a situation |
| `subslots` | (Slot) Immediate subslots of a slot |
| `sum` | (Numbers) Sum of a set of numbers |
| `superclasses` | (Class) Immediate superclasses of a class |
| `supersituations` | (Situation) Immediate supersituations of a situation |
| `superslots` | (Slot) Immediate superslots of a slot |
| `third` | (Set) Third element in a set of values† |
| `unification` | (Set) Unify (using `&`) all the members of a set |

**Notes:**

- † Preservation of slot-value ordering in a set is not guaranteed.
- `instances`, `instances-of` are inertial fluents. `add-list`, `del-list`, `pcs-list`, `ncs-list` are non-inertial fluents. All other built-in slots are non-fluents.

## 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] Peter Clark and Bruce Porter. KM – the knowledge machine: Users manual. Technical report, AI Lab, Univ Texas at Austin, 1999. (http://www.cs.utexas.edu/users/mfkb/km.html).

[5] 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).