From Natural Language to KM Representations
Working Note 24

Peter Clark
Knowledge Systems
Boeing Maths and Computing Technology
peter.e.clark@boeing.com

Abstract

This working note sketches out, in a rather skeletal form, a model of language understanding using the knowledge representation language KM. In this model, KM "prototypes" encode stereotypical expectations about the world, and language understanding involves constructing a scene from prototypes which best matches (a possible interpretation of) the input text. As a starting point towards this, we describe a simplified version of this process where the task is just to find the best match between a single prototype and a possible interpretation of the input.

1 Introduction and Approach

This brief working note sketches out some initial thoughts on how to perform controlled language to KM translation. The approach we describe here treats language understanding primarily as a "scene-building" task, in which the goal is to construct the most coherent "picture" of what the input sentences are trying to describe. Background knowledge plays a key role in this process: scenes are constructed by "plugging together" various fragments of background knowledge which the input text suggests, and common-sense constraints are used to help assess the degree of coherence of various possible scenes which can thus be constructed. The most coherent resulting scene is deemed the best understanding of the input text.

By "scene" and (equivalently) "interpretation", we mean a set of facts which the input text states, implies, or merely suggests. By emphasizing background knowledge, we are adopting a particular philosophical stance on the language understanding task, namely that main content of the interpretation comes from background knowledge, rather than directly from the input text itself. Groesser [Groesser, 1981] (as reported in [Zadrozny and Jensen, 1991]) suggests that ratio of background to stated knowledge in an interpretation is around 8:1, a position we concur with. The role of the input text is thus primarily to guide the system in constructing a scene, rather than to simple state the scene itself. One can think of scene-building as being a bit like doing a jigsaw puzzle, where the background knowledge provides the predefined "pieces" of the puzzle, and the input text provides rough hints about which pieces should be used, and which should connect with which. Another analogy is that the input text is a kind of (ambiguous) "program" describing how to construct a scene from pre-defined pieces.

This approach also contrasts with a view of language understanding solely as a knowledge-poor transformational process, in which interpretation is viewed as applying a sequence of syntactic transformations on an input structure to create an output structure (e.g. transforming "John loves mary" to loves(John,Mary)). Transformational approaches rely more heavily on surface-level features as the main clues for sentence understanding, though in practice, in more sophisticated transformational systems, transformation rules also embody considerable domain knowledge and the distinction between these two approaches becomes more a matter of emphasis.

Viewing language understanding as heavily based on background knowledge is not a new idea, and has been at the heart of many NLU systems, in particular the script-based "story understanding" systems of the '70s such as Frump [DeJong, 1979], Sam [Cullingford, 1977], and Boris [Dyer, 1981], and more recent efforts such as [Zadrozny and Jensen, 1991]. However, while the approach is intuitively appealing, it has also proved difficult to operationalize in a general way. Stumbling blocks include: how to represent background knowledge in the first place; the labor of encoding the vast amount of background knowledge required; and problems of controlling search.

The goal of this working note is not to "solve" this problem, but merely to illustrate how this problem might be addressed in the context of the KM Knowledge Representation language [Clark and Porter, 1999].

2 A Toy Model of the Process

2.1 The Role of Background Knowledge

Intuitively, our background knowledge should describe possible configurations of the world, i.e., "ways in which the world can be." Background knowledge should bring a sense of expectation to the language understanding process.

As an introspective exercise, consider the word "computer" and the expectations that the word conjours up. In my mind's eye I picture a monitor about 18" high, a rectangular tower containing a processor, a keyboard and mouse, sitting on a desk somewhere, etc. This mental picture provides a framework for understanding: for example, if a reference to "the mouse" later appears, I may guess this refers to the mechanical mouse associated with this computer. This of course isn't the only possible mental picture of a computer: other ones that come to mind include a 1970's mainframe; a laptop; etc. It is these mental pictures which guide language understanding: text conjours up pictures, and pictures suggest interpretations of text.

In these examples, some parts of my mental "picture" of a concept are true of all examples of that concept (e.g. all computers have a keyboard, say), while others are incidental (e.g. the computer's color is grey). In fact, what I'm really doing is recalling exemplars of a concept, and a second part of the understanding process involves the ability to adapt or "mutilate" such pictures so that they better fit the narrative (but only mutilate incidental parts). For now, though, in an attempt to create a computational model of this process, we'll avoid this second "adaptation" issue, and just consider "generalized" pictures which only contain universally true facts.

2.2 Representing Background Knowledge using Prototypes

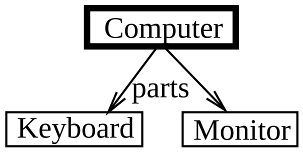
There are several ways such "pictures" can be encoded in KM, but the most convenient is to use KM's prototype mechanism (there are also other ways, which we mention later in Section 4). A KM prototype is a set of ground assertions between anonymous Skolem instances, i.e. objects which denote *some* instance of a class. For example, the prototype:

```
(_ProtoComputer3 has
  (instance-of (Computer))
  (parts (_ProtoKeyboard1 _ProtoMonitor2)))

(_ProtoKeyboard1 has
  (instance-of (Keyboard))
  (part-of (_ProtoComputer3)))

(_ProtoMonitor2 has
  (instance-of (Monitor))
  (part-of (_ProtoComputer3)))
```

describes a computer with a keyboard and a monitor. We can sketch this graph of Skolem instances (the "instance graph") as follows:



In each prototype one instance is labelled as the "root" (here `_ProtoComputer3`). The semantics of prototypes is that, for all instances of the root's most specific class (here, `Computer`), the relationships described in the prototype hold. In this case, as an instance of `Computer` is the root, the prototype asserts that "all computers have a keyboard and monitor." A prototype is thus a giant "forall...exists..." statement.

While we could also represent the same knowledge as KM rules (e.g. `(every Computer has (parts ...))`), prototypes turn out to be very convenient for two reasons. First, they have a very simple syntactic form, and second there is a clear boundary on what they contain. On this latter point, a prototype does not have to encode *every* fact about a concept, but rather just some subset of facts about it, as selected by the Knowledge Engineer when encoding the prototype in the first place. Multiple prototypes can be used to define a single concept, with each prototype encoding a different piece of the representation. This allows us to factor up a complex concept representation into more modular pieces.

2.3 Input Text suggests Background Knowledge

A scene can be constructed from prototypes by (i) identifying which prototypes to use (ii) cloning them (i.e., renaming the Skolem constant names), and (iii) connecting them, by asserting equalities and other relationships between the entities they contain. Language interpretation is then a search process, searching for the most "coherent" such scene.

The first of these tasks, and the only one we examine in this working note, is that of identifying prototypes which are suggested by the input text. This suggestion might work as follows:

1. If the input text uses a word, and one sense of that word is a concept used in a prototype, then that prototype is potentially applicable (it has been "activated").

   For example, the word "computer" suggests various prototypes that use the concept `Computer` (including its specializations). These include prototypes *of* computers (e.g., `Computer`, `Laptop-Computer`, `Desktop-Computer`, etc.), and prototypes *mentioning* a computer somewhere (e.g. `Office`, which includes a computer sitting on a desk, and `Business-Trip`, which involves an employee travelling carrying a computer with him/her).

Ambiguous words would suggest prototypes reflecting the different word senses. For example, the word "switch" suggests prototypes for the concepts of `Electromechanical-Switch` and `Switch-Stick` might be applicable. Semantic preprocessing can obviously help here in eliminating some of these alternatives up front.

2. If the input text makes an *assertion* also contained in a prototype, then that prototype is potentially applicable.

   For example, if the text states "the person was carrying a computer", and a prototype for `Business-Trip` includes a person carring a computer, then the `Business-Trip` prototype is suggested by the text.

2.4 Assessing Coherence

For each suggested prototype, the computer then needs to assess how well it accounts for the input text. In our trivial computational model, we do this simply by seeing how many of the prototype's concepts have some word in the input text corresponding to them. The higher the number of such "matches", the more coherent the prototype is as an interpretation of the input text.

3 A Worked Example

3.1 Task and Initial Knowledge

Here is a (very) toy worked example of this process. Consider interpreting the single input sentence:

> "He pushed the switch."

A few of the inferences we would like the computer to draw when interpreting this are:

- Although the word "switch" is ambiguous, it here (probably) means electrical switch, and thus there's presumably some electrical circuit which the switch is part of etc.

- The switch is probably a push-button switch, as opposed to a toggle switch, rotating switch etc.

We assume no semantic preprocessing of the text, although clearly this would be extremely helpful at reducing some of ambiguity and intepretation.

Let us also assume we have background knowledge prototypes as shown in Figure 1, encoding knowledge about different types of switches. The actual KM code for these prototypes is shown in Appendix A.

We also consider a (many-to-many) mapping between words and concepts as shown below (and in Appendix A). We can think of this as a poor man's (or rather, bankrupt man's) lexicon, mapping words to word senses:

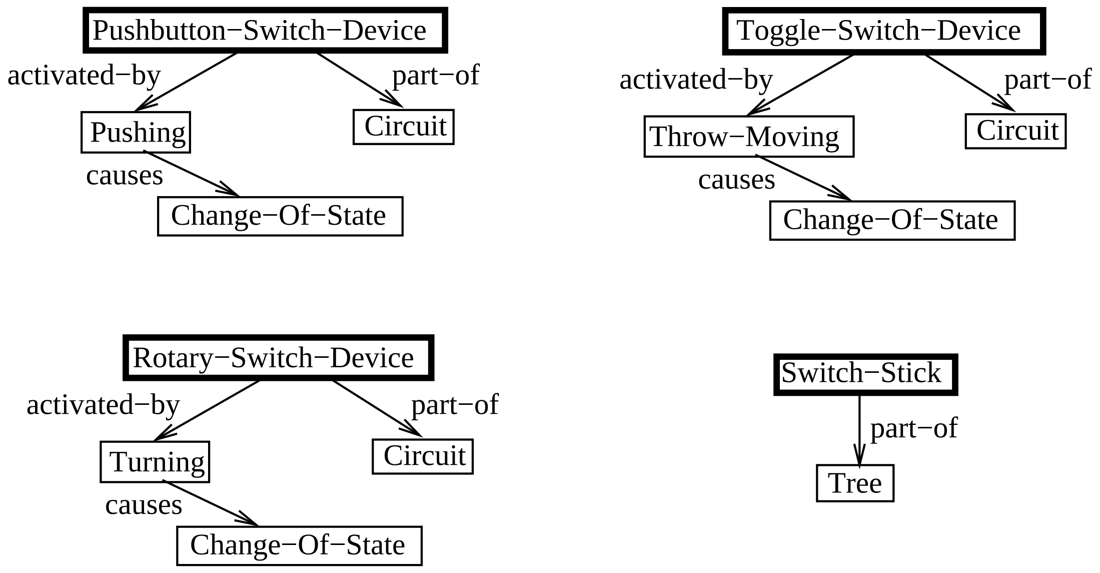


Figure 1: Prototypes encoding knowledge about different types of "switch" in the toy KB.

Words	**Concepts**
"switch"	`Changing`, `Switch-Device`, `Switch-Stick`
"push", "pushed"	`Pushing`
"throw", "threw"	`Throw-Propelling`, `Throw-Moving`
"turn", "turned"	`Turning`

3.2 Interpreting the Input Text

As a simplification of the process, we consider the task to be to find the *single* prototype best matching the input, i.e. we do not here consider selecting multiple prototypes. This makes the task straightforward:

- For each word in the input, find the concept(s) which it might refer to (using the toy lexicon above). Note that we perform no syntactic processing whatsoever, thus word order is irrelevant. This is obviously a major deficiency!
- Thus, for the input sentence, identify its possible *interpretations*, where an interpretation is a particular choice of word sense for each word in the sentence.
- For each prototype in the KB, assess the degree of match between the prototype and each interpretation, where degree of match is simply the number of concepts in the interpretation also used in that prototype. Figure 2 illustrates one possible interpretation of the sentence "he pushed the switch", which has two matches.

A trace of a little brute-force program performing this search is as follows:

```
USER(20): (doit)
;;; Enter initial sentence
Enter sentence: he pushed the switch

;;; There are three alternative interpretations of the words here (each line shows a
;;; different interpretation)
```

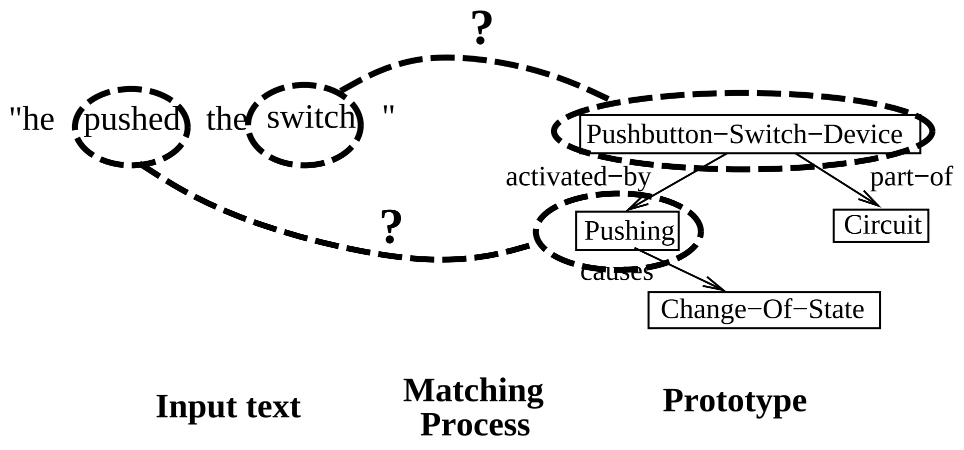


Figure 2: Input text is interpreted by matching it with a predefined possible configuration of the world, represented using a KM prototype. The overall task is to then find the best such interpretation.

```
interpretations =
  (("pushed" Pushing) ("switch" Changing))
  (("pushed" Pushing) ("switch" Switch-Stick))
  (("pushed" Pushing) ("switch" Switch-Device))

;;; We exhaustively assess the degree of match between each interpretation
;;; and each prototype in the KB (Search space size = 3 * 4 = 12). Below
;;; lists the prototype, and the degree of match with the input interpretation,
;;; and the word–concept–prototype-elements matches found. A “?” denotes
;;; that no match was found.
Prototypes fitting the data:
  Rotary-Switch-Device (score 0)
     [for interpretation (("pushed" Pushing ?) ("switch" Changing ?))]
  Rotary-Switch-Device (score 0)
     [for interpretation (("pushed" Pushing ?) ("switch" Switch-Stick ?))]
  Rotary-Switch-Device (score 1)
     [for interpretation (("pushed" Pushing ?)
          ("switch" Rotary-Switch-Device _ProtoRotary-Switch-Device9))]
  Toggle-Switch-Device (score 0)
     [for interpretation (("pushed" Pushing ?) ("switch" Changing ?))]
  Toggle-Switch-Device (score 0)
     [for interpretation (("pushed" Pushing ?) ("switch" Switch-Stick ?))]
  Toggle-Switch-Device (score 1)
     [for interpretation (("pushed" Pushing ?)
          ("switch" Toggle-Switch-Device _ProtoToggle-Switch-Device5))]
  Pushbutton-Switch-Device (score 1)
     [for interpretation (("pushed" Pushing _ProtoPushing2) ("switch" Changing ?))]
  Pushbutton-Switch-Device (score 1)
     [for interpretation (("pushed" Pushing _ProtoPushing2)
          ("switch" Switch-Stick ?))]
  Pushbutton-Switch-Device (score 2)
     [for interpretation (("pushed" Pushing _ProtoPushing2)
          ("switch" Pushbutton-Switch-Device _ProtoPushbutton-Switch-Device1))]
  Switch-Stick (score 0)
```

```
      [for interpretation (("pushed" Pushing ?) ("switch" Changing ?))]
    Switch-Stick (score 1)
      [for interpretation (("pushed" Pushing ?)
                           ("switch" Switch-Stick _ProtoSwitch-Stick13))]
    Switch-Stick (score 0)
      [for interpretation (("pushed" Pushing ?) ("switch" Switch-Device ?))]

;;; Pick out the match with the highest score...
Best interpretation:
  Pushbutton-Switch-Device (score 2)
      [for interpretation (("pushed" Pushing _ProtoPushing2)
           ("switch" Pushbutton-Switch-Device _ProtoPushbutton-Switch-Device1))]
Therefore:
        "pushed" -> Pushing
        "switch" -> Pushbutton-Switch-Device

;;; Given this interpretation, the prototype supplies the following
;;; additional assertions ("expectations") also:
Additional inferences:
   Pushbutton-Switch-Device activated-by (Pushing)
   Pushbutton-Switch-Device part-of (Circuit)
   Pushing causes (Change-Of-State)
   Pushing activated-by-of (Pushbutton-Switch-Device)
   Change-Of-State causes-of (Pushing)
   Circuit parts (Pushbutton-Switch-Device)
```

What has happened here? The system has taken the sentence "he pushed the switch" and found that the best-matching prototype is the one describing a pushbutton switch. This is the best match because that prototype contains both a `Pushing` and a `Pushbutton-Switch-Device`, which are possible interpretations of the input words "pushed" and "switch" respectively. Thus, this prototype accounts for two of the four input words, better than any other prototype.

As a result of this match, the system can conclude that "pushed" probably means the concept `Pushing` and "switch" probably means the concept `Pushbutton-Switch-Device`. It can also conclude (from the prototype) that the pushing means pushing of the switch, that the switch is probably part of a circuit, and that the pushing will probably result in a change of state. These additional inferences are supplied by the matching prototype.

4 Prototypes and Other Mechanisms

We have suggested KM prototypes as a means of representing background expectations. However, the key property that we want is not "prototypes" per se, but rather a "graph of instances" representation which can be matched with the input data. The main benefit of this form is its syntactic simplicity, the representation containing only ground relationships between Skolem instances. Prototypes are just one convenient way of encoding such graphs in KM. In fact, the semantics of prototypes (as "forall...exists..." statements) are not used in the simple algorithm above; all the algorithm cares about is that each graph denotes a plausible configuration of the world.

An alternative to using prototypes would be to generate similar "instance graphs" from a regular KM KB. We will call this process *actualizing* a concept in the KB. Consider a trivial representation of a computer using a standard KM `(every ...)` expression:

```
(every Computer has
  (parts ((a Keyboard) (a Monitor))))
```

The KB should thus "expect" a computer to have a keyboard and a monitor. But working directly with this frame data structure is complex, and it would be easier if it was transformed into an "instance graph" form. We can do this simply by the following steps:

1. Create an instance of `Computer`
2. For all slots on computer, compute their values.
3. Apply this procedure recursively (and with some depth limit) to find the slots of those values.

The result will be an *example* of a computer in KM's memory, generated from the general rules in the KB:

```
(_Computer3 has
  (instance-of (Computer))
  (parts (_Keyboard1 _Monitor2)))

(_Keyboard1 has
  (instance-of (Keyboard))
  (part-of (_Computer3)))

(_Monitor2 has
  (instance-of (Monitor))
  (part-of (_Computer3)))
```

This set of ground assertions can be used in exactly the same way as a prototype. We only need to add suitable bookkeeping to note that the root `_Computer3` is an "actualized example" of a KB concept, and that `_Keyboard1` and `_Monitor2` are part of it, so that the text interpretation algorithm can find this example during search.

This process of actualizing KB concepts has already been used in the DARPA RKF project, for a different purpose of supporting analogical reasoning. Again, the motivation is that it is easier to manipulate these "instance graphs" than work with the general rules in the KB directly.

A final alternative is simply enter such examples manually into the computer, without any implied universal quantification. These examples then denote cases rather than general assertions. Handling cases like this requires more sophisticated reasoning, however, as a mechanism for modifying and adopting such examples would be required, as (unlike prototypes and actualized concepts) hand-created examples may also include incidental as well as logically implied propositions.

5 Summary

This working note has sketched out, in a very skeletal form, the start of a model of language understanding using KM. Prototypes encode stereotypical expectations about the world, and language understanding involves constructing a scene from prototypes which best matches (a possible interpretation of) the input text. We have presented a simplified version of this where the task is simply to find the best match between a single prototype and a possible interpretation of the input, and hope that further generalization of this process will be forthcoming.

References

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

[Cullingford, 1977] Cullingford, R. E. (1977). Controlling inference in story understanding. In *IJCAI-77*, page 17.

[DeJong, 1979] DeJong, G. (1979). Prediction and substantiation: two processes that comprise understanding. In *IJCAI-79*, pages 217–222.

[Dyer, 1981] Dyer, M. G. (1981). $RESTAURANT revisited, or 'lunch with boris'. In *IJCAI-81*, pages 234–236.

[Groesser, 1981] Groesser, A. C. (1981). *Prose Comprehension Beyond the Word*. Springer, NY.

[Zadrozny and Jensen, 1991] Zadrozny, W. and Jensen, K. (1991). Semantics of paragraphs. *Computational Linguistics*, 17(2):171–209.

Appendix A: KM Code for the Example

```
;;; Define the parts/part-of inverse relationship
(parts has
  (instance-of (Slot))
  (inverse (part-of)))

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

;;; Actions
(Changing has (superclasses (Action)) (words ("switch")))
(Pushing  has (superclasses (Action)) (words ("push" "pushed")))
(Throw-Propelling has (superclasses (Action)) (words ("throw" "threw")))
(Throw-Moving has (superclasses (Action)) (words ("throw" "threw")))
(Turning  has (superclasses (Action)) (words ("turn" "turned")))

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

;;; Electrical Switch Devices
(Switch-Device has
  (words ("switch")))

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

(Pushbutton-Switch-Device has (superclasses (Switch-Device)))

(a-prototype Pushbutton-Switch-Device)
((the Switch-Device) has
   (activated-by ((a Pushing with (causes ((a Change-Of-State)))))))
((the Switch-Device) has (part-of ((a Circuit))))
(end-prototype)

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

(Toggle-Switch-Device has (superclasses (Switch-Device)))

(a-prototype Toggle-Switch-Device)
((the Switch-Device) has
   (activated-by ((a Throw-Moving with (causes ((a Change-Of-State)))))))
((the Switch-Device) has (part-of ((a Circuit))))
(end-prototype)

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

(Rotary-Switch-Device has (superclasses (Switch-Device)))

(a-prototype Rotary-Switch-Device)
((the Switch-Device) has
   (activated-by ((a Turning with (causes ((a Change-Of-State)))))))
((the Switch-Device) has (part-of ((a Circuit))))
(end-prototype)

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

;;; Another meaning of the noun "switch"
```

```
(Switch-Stick has
  (superclasses (Stick))
  (words ("switch")))

(a-prototype Switch-Stick)
((the Switch-Stick) has (part-of ((a Tree))))
(end-prototype)
```

Appendix B: Notes on KM Prototypes

For details on how to create prototypes in KM, see the KM User Manual [Clark and Porter, 1999]. Here are some additional notes on the internal representation of prototypes, based on an example. First, this is how a little prototype of a `Computer` would be defined:

```
KM> (a-prototype Computer)

[prototype-mode] KM> ((the Computer) has (parts ((a Keyboard) (a Monitor))))

[prototype-mode] KM> (end-prototype)
```

Here is the internal form of that prototype, generated by KM from the above user input:

```
KM> (write-kb)                        ; (just showing selected parts below)

(Computer has
  (instances (_ProtoComputer1))
  (prototypes (_ProtoComputer1)))

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

(_ProtoComputer1 has
  (instance-of (Computer))
  (prototype-of (Computer))
  (definition ('(a Computer)))
  (protoparts (_ProtoComputer1
               _ProtoKeyboard2
               _ProtoMonitor3))
  (protopart-of (_ProtoComputer1))
  (parts (_ProtoKeyboard2
          _ProtoMonitor3)))

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

(_ProtoKeyboard2 has
  (instance-of (Keyboard))
  (protopart-of (_ProtoComputer1))
  (parts-of (_ProtoComputer1)))

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

(_ProtoMonitor3 has
  (instance-of (Monitor))
  (protopart-of (_ProtoComputer1))
  (parts-of (_ProtoComputer1)))

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

(Keyboard has (instances (_ProtoKeyboard2)))

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

(Monitor has (instances (_ProtoMonitor3)))
```

The prototype consists of a set of Skolem instances, and a set of binary relations between them. One of these Skolem instances is privileged, and is called the *root* of the prototype. The prototype graph is considered to hold for *all* instances of matching the definition of that root.

The set of instances in the prototype are called its *protoparts*, and can be found by looking on the `protoparts` slot of the prototype root (see above).

For working with prototypes, the following queries may be useful, shown here issued from the Lisp prompt (see the KM User Manual for more on calling KM with a `(km ...)` call):

```
;;; Find all the prototypes of the class Computer
USER(4): (km '#$(the prototypes of Computer) :fail-mode 'fail)
(_ProtoComputer1)

;;; Find all the parts of _ProtoComputer1
USER(5) (km '#$(the protoparts of _ProtoComputer1) :fail-mode 'fail)
(_ProtoComputer1 _ProtoKeyboard2 _ProtoMonitor3)

;;; Find the slots and values of _ProtoComputer1. (Below, all these slots are
;;; bookkeeping except for the parts slot):
USER(5) (get-slotsvals '#$_ProtoComputer1)
((instance-of (Computer))
 (prototype-of (Computer))
 (definition ('(a Computer)))
 (protoparts (_ProtoComputer1 _ProtoKeyboard2 _ProtoMonitor3))
 (protopart-of (_ProtoComputer1))
 (parts (_ProtoKeyboard2 _ProtoMonitor3)))
```