Current State of the Noun Definition Algorithm and Associated Demonstrations

Scott Napieralski
stn2@cse.buffalo.edu

May 1, 2003

Abstract

This document describes work on the Contextual Vocabulary Acquisition project that was conducted during the Spring 2003 semester under the supervision of Dr. William Rapaport. The majority of the work during this semester was focused on updating, improving and annotating the various demonstration programs that were originally created by Karen Ehrlich. These changes will be described in detail. In addition, a short examination of the verb algorithm was performed. This work will also be described.

1 Introduction

The work described in this paper was performed as part of the Contextual Vocabulary Acquisition project. Researchers from the University at Buffalo Departments of Computer Science & Engineering and Learning & Instruction are cooperating to develop a greater understanding of the human process of learning new vocabulary from context clues. In order to improve our understanding and test our theories, we have made use of a computational algorithm that attempts to model the human process of defining an unknown noun based on context.

The noun definition algorithm implements a theory of how an unknown word can be learned based on the passages of text in which the unknown word occurs. The information that is contained in a passage is first encoded as a SNePS network [Shapiro 1999]. Then, when asked to provide a definition for the unknown word the algorithm searches the SNePS network for certain types of information. When all the relevant information in the network has been found, the algorithm reports it as a definition for the unknown word.

The major goal of the research performed this semester was to complete work on the demonstration programs that was begun during the previous semester [Napieralski 2002B]. There are several demonstration programs derived from Karen Ehrlich's dissertation [Ehrlich 1995] that illustrate the various capabilities of the noun algorithm. In order to similarly demonstrate the capabilities of the improved noun definition algorithm [Napieralski 2002A], these programs required some updates and modifications. Specifically the "brachet", "cat", and "hackney" demos have each been updated, so that they generate definitions that are at least as complete as the definitions created by the original demos in combination with Ehrlich's original noun definition algorithm [Ehrlich 1995].

2 Changes to the Demos

The first task undertaken this semester was to complete the work on the "brachet" demo that was left incomplete after the fall semester. This consisted of thoroughly annotating the demo and changing the order in which the information was entered. Previously, the information in the background knowledge file had been in nearly random order. In the updated file, semantically similar facts and rules are grouped together. This does not have any effect on SNePS or the algorithm, but it does make the file easier for a human to read and understand. The completed demo was then sent to Fran Johnson to be included in the set of standard SNePS demonstrations.

The next task was to examine the "cat" demo and attempt to update it so that it would produce a definition that was at least as good at the definition given by Ehrlich's original demonstration. The major problem with this demo was the difficulty of inferring that cats are mammals after we learn that Pyewacket bears kittens. At this point in the demonstration, we know that Pyewacket is a mammal and Pyewacket is a cat and we want to infer that cats are mammals. In order to make this inference, a new rule (figure 1) was developed (with the help of Dr. Shapiro). The rule is as follows: if X is a member of some class Y and X is a member of some class Z and Y is an unknown word and Z is not (known to be) a subclass of Y, then presumably Y is a subclass of Z. In order to make the part of the rule that says "Z is not (known to be) a subclass of Y" work we used SNeRE, the SNePS Rational Engine [Shapiro 1999]. Although this makes

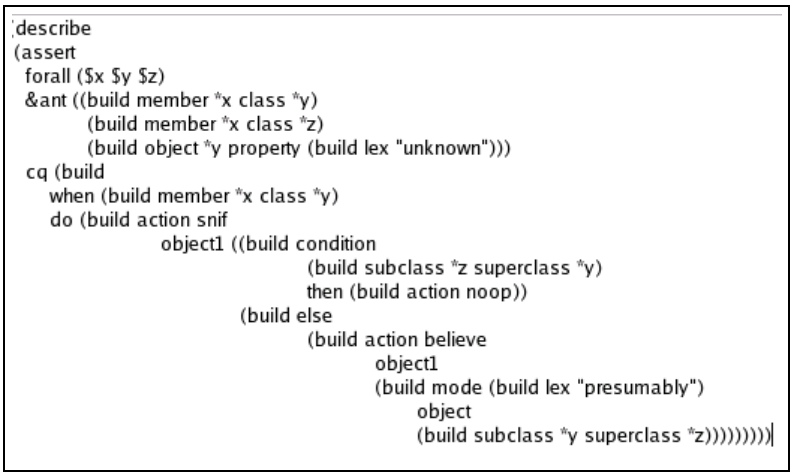

Figure 1: This rule allows us to infer that cats are mammals.

for a more complicated rule, it does not change the way that the demo is invoked or used. In addition, to prevent degenerate situations where SNePS could infer that X is a subclass of Y and Y is a subclass of X, we added the antecedent that says "Y is an unknown word". By marking the target word as unknown, we are modeling the cognitive agent's awareness that it does has encountered and unknown word and that it is trying to compute a definition. In order to trigger this antecedent, it was necessary to add some information to the demonstration. At the beginning of the demo, "cat" is now explicitly marked as a unknown word. This marking is accomplished by using the "object-property" case frame [Shapiro 1996] where "cat" is the object and its property "unknown".

The final noun demo that has been modified is the "hackney" demo. This demo required the fewest modifications, it immediately provided the same definition using the new noun definition algorithm that it had provided using Ehrlich's original noun definition algorithm. Unlike the other demos, changes that were made to this demo were designed to exploit some information that the original demo had not picked up, rather than to fix errors or incomplete definitions.

Early in the demo we learn that King Arthur gets on a hackney, then a short time later we learn that he has jumped off of a horse. Based on this sequence of events, we believed the system should conclude that hackneys are horses. In order to achieve this goal, the following rules were added to the background knowledge. The first rule states that if X leaps onto Y and sometime later X leaps off of Z then Y and Z are equivalent. The second rule says that if two individuals are equivalent and they are each a member

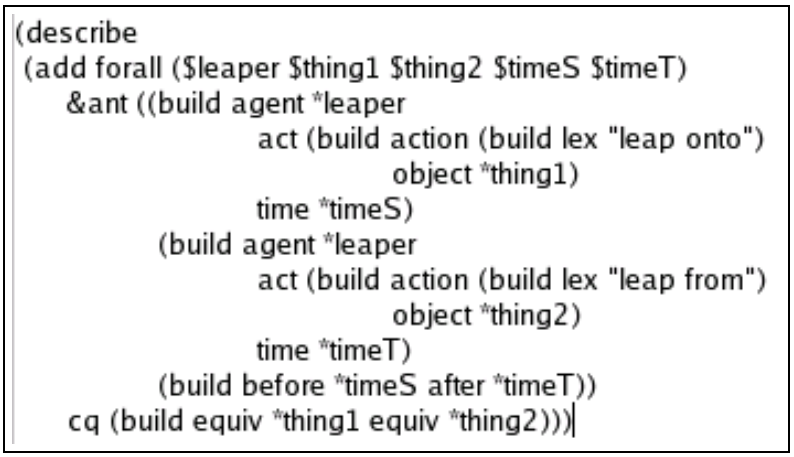

Figure 2: If something leaps onto Y and then leaps off of Z, Y and Z are equivalent.

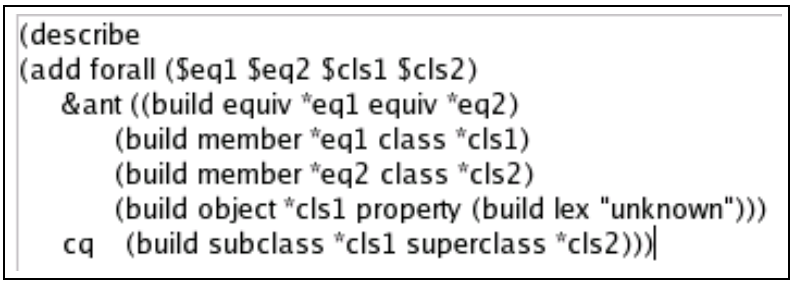

Figure 3: This rule allows us to infer that cats are mammals.

of some class where the class is unknown, then the unknown class is a subclass of the known class. The latter rule requires that "hackney" be marked as an unknown word, just as "cat" was previously marked as "unknown".

Unfortunately, these rules did not provide the results we were seeking without further modification of the demo. When the system was told that King Arthur got off of a horse (using the "add" command) [Shapiro 1999] the forward inference procedure failed to conclude that the thing King Arthur got on and the thing that he got off of were equivalent. In order to get the system to infer this information, it was necessary to issue the "clear-infer" [Shapiro 1999] command immediately before adding the knowledge that King Arthur jumped off a horse. With the addition of a "clear-infer" the system was able to infer the necessary information.

Currently, the algorithm correctly reports that a hackney is a type of horse, but it does not report this information as quickly as expected. After we learn that King Arthur jumped off a horse, horse only appears in the definition as a possible class inclusion. The algorithm does not report the fact that a hackney is

definitely a type of horse until it is asked to provide a definition three more times. At the conclusion of my work this semester, the reason for this behavior is still unknown. I suspect that it might be caused when "deduce" commands [Shapiro 1999] embedded in the algorithm find the information at a later time, but I cannot be sure. The next researcher to work on this project should investigate this matter, both to resolve the problem and as an excellent way to familiarize himself/herself with the noun algorithm.

3 The Verb Algorithm

The other major task that I performed this semester was an examination of the verb algorithm that was created during the summer of 2002 by Justin Del Vecchio [Del Vecchio 2002]. Unfortunately, the spring semester came to a close before I could complete a detailed examination of how the algorithm operates and compare it to Ehrlich's original verb definition algorithm [Ehrlich 1995].

During the relatively short period of time that I did work on the verb algorithm, several changes were made. In order to get the verb algorithm to correctly run on the UB CSE systems under ACL 6.2, several minor modifications were necessary. These changes were all syntactic in nature, such as adding empty argument lists to functions that take no arguments. The only other change to the Del Vecchio's verb algorithm was that the trace level argument to the main function (defineVerb) was made optional, with the default set to no tracing.

In order to test the verb algorithm, it was necessary to create a demonstration program for a verb. I chose to recreate the "joust" demo described in Ehrlich's dissertation [Ehrlich 1995]. In recreating this demo, I attempted to follow Ehrlich's demonstration as closely as possible. Despite this fact, several changes to the "joust" demo were made, but they all were minor changes. Specifically, the case frames were changed to conform to the set of case frames expected by the most recent versions of the noun and verb definition algorithms. The "joust" demo was tested using both the noun and verb algorithms, but there was no time for a detailed examination of this demo so no major changes to the demo were performed.

4 Future Work

The immediate next step for my successor should be to investigate the strange behavior of the "hackney" demo that is described above. Specifically, the next researcher should attempt to determine why using "clear-infer" produces better results and why the algorithm only reports "horse" as a superclass of "hackney" after the third request for a definition.

Another issue that the next researcher may wish to investigate is the possibility of merging all the background knowledge files. Currently, each demonstration program has its own set of background knowledge (contained in files with the extension ".base"). Ehrlich's original intent was to have a single set of background knowledge that was used for all of the demos. Fortunately, all of the background files are fairly similar. It should be possible to merge them into a single file containing all the knowledge that is common to each of the background files plus all the background information that is unique to each demo. However, if such a merger is attempted, a thorough examination of each of the demos will be required to insure that the change has had no unexpected effects. If my experience working on this project has taught me anything, it is that even the most seemingly benign changes can wreak havoc in a complex system such as SNePS.

Once work on the noun demonstrations has been successfully completed, the researcher should investigate the verb algorithms, both Del Vecchio's version and Ehrlich's version. Neither of these algorithms has been nearly as well researched and implemented as the noun definition algorithm. In the future, significant effort should be devoted to creating a verb definition algorithm that is at least as complete as the noun definition algorithm. After the next investigator has developed a thorough understand of the verb algorithms by examining them and attempting to develop new demos for them, a longer term goal should be to improve Del Vecchio's algorithm. This can probably accomplished by incorporating some of Ehrlich's old ideas and implementing some new techniques.

References

[Del Vecchio 2002] Del Vecchio, J. (2002), "Summer Research - 2002 - Verb Algorithm", [http://www.cse.buffalo.edu/ jmdv/summer2002-verbalgorithm-jmdv/].

[Ehrlich 1995] Ehrlich, K. (1995), "Automatic Vocabulary Expansion through Narrative Context", *SUNY Buffalo Computer Science Technical Report 95-09.*

[Napieralski 2002A] Napieralski, S. (2002), "An Enhanced Noun Definition Algorithm" [http://www.cse.buffalo.edu/ stn2/cva/summer02/summer-report.pdf].

[Napieralski 2002B] Napieralski, S. (2002), "Progress of the Noun Definition Algorithm During the Fall 2002 Semester", [http://www.cse.buffalo.edu/ stn2/cva/fall02/reportFall02.pdf].

[Shapiro 1996] Shapiro, S., et. al. (1996). "A Dictionary of SNePS Case Frames" [http://www.cse.buffalo.edu/sneps/Manuals/dictionary.pdf].

[Shapiro 1999] Shapiro, S. (1999), "SNePS 2.5 User's Manual" [http://www.cse.buffalo.edu/sneps/Manuals/manual25.ps].

A The Noun Algorithm

```
(in-package :snepsul)

(defvar *dmode* nil
  "Indictates whether algorithm should operate in definition mode (t) which
    uses Ehrlich's logic to decide which information should be reported and
    which infomation should be ignored, or teaching mode (nil) which reports
    all information that can be found.")

(defstruct ndefn
  "A structure to store and report definitions of nouns"
  noun
  classInclusions
  probableClassInclusions
  possibleClassInclusions
  structuralElements
  probableStructuralElements
  possibleStructuralElements
  actions
  probableActions
  possibleActions
  properties
  probableProperties
  possibleProperties
  owners
  synonyms
  possibleSynonyms
  agents
  spatial
  namedIndividuals)

;;;-------------------------------------------------------------------------
;;;
;;;      function: defineNoun
;;;                                                        created: stn 2002
;;;-------------------------------------------------------------------------
(defun defineNoun (noun &optional (lexicographicMode t) (traceLevel -1))
  "Generates a definition for 'noun'. If the optional argument
    lexicographicMode is t then Ehrlich's theory will be used to exclude
    some information from the definition, else all info will be reported.
    If the optional argument traceLevel is specified tracing/debugging will
    be enabled. The values of traceLevel are 0-4 where 0 means no
```

```
    tracing and 4 means trace all functions."
  ;; the default for traceLevel is -1 so that any tracing set up manually
  ;; by the user will not be overridden by the program when the optional
  ;; argument traceLevel is not specified.
  (setTraceLevel traceLevel)
  (setq *dmode* lexicographicMode)
  ;; get the requested definition and print it in human readable format.
  (prettyPrintDef (if lexicographicMode
                      (defineNounLexicographic noun)
                    (defineNounTeaching noun))))

;;;-------------------------------------------------------------------------
;;;
;;;     function: defineNounLexicographic
;;;                                                        created: stn 2002
;;;-------------------------------------------------------------------------
(defun defineNounLexicographic (noun)
  "Makes a list of information that is known about the noun and reports only
    the information that is deemed relevant according to Ehrlich's theory."
  (let (definition)
    ;; get all the info
    (setf definition (defineNounTeaching noun))
    ;; Now examine all the info and eliminate parts that Ehrlich's theory
    ;; says are unnecessary.

    ;; if there are class inclusions, don't report probable class inclusions
    (if (ndefn-classInclusions definition)
        (or (setf (ndefn-probableClassInclusions definition) nil)
            (setf (ndefn-possibleClassInclusions definition) nil)))
    ;; if there are probable class inclusions, don't report possible
    ;;   class inclusions
    (if (ndefn-probableClassInclusions definition)
        (setf (ndefn-possibleClassInclusions definition) nil))
    ;; if there are structural elements don't report probable or possible
    ;;  struct. elems.
    (if (ndefn-structuralElements definition)
        ;; using "or" is just a way to make sure that both setf statements
        ;;   get evaluated
        (or (setf (ndefn-probableStructuralElements definition) nil)
            (setf (ndefn-possibleStructuralElements definition) nil)))
    ;; if there are probable structural elements don't report possible
    ;;  structural elements
    (if (ndefn-probableStructuralElements definition)
        (setf (ndefn-possibleStructuralElements definition) nil))
    ;; if there are actions don't report probable or possible actions
    (if (ndefn-actions definition)
        (or (setf (ndefn-probableActions definition) nil)
            (setf (ndefn-possibleActions definition) nil)))
    ;; if there are probable actions don't report possible actions
    (if (ndefn-probableActions definition)
        (setf (ndefn-possibleActions definition) nil))
    ;; if there are any type of actions, don't report agents
    (if (or (ndefn-actions definition)
            (ndefn-probableActions definition)
            (ndefn-possibleActions definition))
        (setf (ndefn-agents definition) nil))
    ;; if there are properties, don't report probable or possible properties
    (if (ndefn-properties definition)
```

```
        (or (setf (ndefn-probableProperties definition) nil)
            (setf (ndefn-possibleProperties definition) nil)))
    ;; if there are probable properties, don't report possible properties
    (if (ndefn-probableProperties definition)
        (setf (ndefn-possibleProperties definition) nil))
    ;; if there are class inclusions or probable class inclusions,
    ;;   don't report named individuals
    (if (or (ndefn-classInclusions definition)
            (ndefn-probableClassInclusions definition))
        (setf (ndefn-namedIndividuals definition) nil))

    ;; now return the revised definition
    definition
    ))

;;;-------------------------------------------------------------------------
;;;
;;;     function: defineNounTeaching
;;;                                                        created: stn 2002
;;;-------------------------------------------------------------------------
(defun defineNounTeaching (noun)
  "Makes a list of all information that is known about the noun.
    This information makes up the definition."
  (let (definition)
    ;; get a new instance of the structure ndefn
    (setf definition (make-ndefn))
    ;; populate the fields of the definition structure
    ;; the noun itself
    (setf (ndefn-noun definition) noun)
    ;; class inclusions
    (setf (ndefn-classInclusions definition) (findClassInclusions noun))
    ;; probable (with 'mode presumably') class inclusions
    (setf (ndefn-probableClassInclusions definition)
      (findProbableClassInclusions noun))
    ;; possible class inclusions
    (setf (ndefn-possibleClassInclusions definition)
      (classFilter
       (findPossibleClassInclusions noun
                (append (ndefn-classInclusions definition)
                        (ndefn-probableClassInclusions definition)))
       noun))
    ;; structure
    (setf (ndefn-structuralElements definition) (findStructure noun))
    ;; probable structure
    (setf (ndefn-probableStructuralElements definition)
      (findProbableStructure noun))
    ;; possible structure
    (setf (ndefn-possibleStructuralElements definition)
      (findPossibleStructure noun (ndefn-classInclusions definition)))
    ;; properties
    (setf (ndefn-properties definition) (findProperties noun))
    ;; probable properties
    (setf (ndefn-probableProperties definition) (findProbableProperties noun))
    ;; possible properties
    (setf (ndefn-possibleProperties definition) (findPossibleProperties noun))
    ;; owners
    (setf (ndefn-owners definition) (findOwners noun))
    ;; spatial information
```

```
    (setf (ndefn-spatial definition) (findSpatial noun))
    ;; synonyms
    (setf (ndefn-synonyms definition) (findSynonyms noun))
    ;; possible synonyms
    (setf (ndefn-possibleSynonyms definition)
      (findPossibleSynonyms noun
                            (union (ndefn-structuralElements definition)
                                   (ndefn-probableStructuralElements definition
))
                            (union (ndefn-classInclusions definition)
                                   (ndefn-probableClassInclusions definition))
                            (ndefn-owners definition)
                            (ndefn-synonyms definition)))
    ;; agents who act on 'noun's
    (setf (ndefn-agents definition) (findAgents noun))
    ;; names of specific 'noun's
    (setf (ndefn-namedIndividuals definition) (findNamedIndividuals noun))
    ;; actions
    (setf (ndefn-actions definition) (act_filter (findActions noun) noun))
    ;; probable actions
    (setf (ndefn-probableActions definition)
      (act_filter (findProbableActions noun) noun))
    ;; possible actions
    (setf (ndefn-possibleActions definition) (findPossibleActions noun))

    ;; we are done using it, so we can run class filter on class inclusion info
    (setf (ndefn-classInclusions definition)
      (classFilter (ndefn-classInclusions definition) noun))
    (setf (ndefn-probableClassInclusions definition)
      (classFilter (ndefn-probableClassInclusions definition) noun))

    ;; return the definition
    definition))

;;;-------------------------------------------------------------------------
;;;
;;;     function: traceLevel
;;;     input:    An integer 0-4 representing the amount of tracing info that
;;;               should be given:
;;;                 0 - no tracing
;;;                 1 - trace only definition function (defineNoun)
;;;                 2 - trace different definition types (lexicographic,
;;;                                                       teaching)
;;;                 3 - trace top level info finding functions
;;;                 4 - trace all info finding functions.
;;;                                                       created: stn 2002
;;;-------------------------------------------------------------------------
(defun setTraceLevel (level)
  (case level
    (0 (untrace defineNoun defineNounTeaching defineNounLexicographic
                structureOfAll findStructure indiv_struct
                findProbableStructure struct-rule struct-presume
                findPossibleStructure
                struct-indiv findClassInclusions findPossibleClassInclusions
                class-indiv findProbableClassInclusions class-sub-sup
                class-rule findActions findProbableActions findPossibleActions
                act-object-rule act-object-&-rule act-object-presum-rule
                act-object-presum-&-rule act-object-noun
```

```
            obj-act-indiv obj-act-presume-&-rule obj-act-presum-rule
            obj-act-&-rule obj-act-rule
            findProperties findProbableProperties
            findPossibleProperties prop-rule
            prop-presume prop-indiv findOwners owner-rel
            owner-poss rel-for-owner
            syn-sub-sup syn-syn findSynonyms findPossibleSynonyms
            eliminateDissimilarClasses similarSuperclassesp noAntonymsp
            antonymp eliminateDissimilarStructure similarStructurep
            eliminateDissimilarOwners findSpatial
            similarOwnersp removeElement findNamedIndividuals named-indiv
            findAgents agent-object action-object prop-relation-1
            prop-& prop-&-relation-1 prop-&-relation-1-presume
            prop-relation-1-presume prop-&-presume prop-relation-2
            prop-&-relation-2 prop-relation-2-presume
            prop-&-relation-2-presume prop-indiv prop-relation-1-indiv
            prop-relation-2-indiv obj-rel-1 obj-&-rel-1 obj-rel-2
            obj-&-rel-2 obj-rel-1-presume obj-&-rel-1-presume
            obj-rel-2-presume obj-&-rel-2-presume obj-rel-1-indiv
            obj-rel-2-indiv loc-prop loc-cls loc-str loc-act-obj
            loc-rel loc-own loc-prop-cat loc-cls-cat loc-str-cat
            loc-act-obj-cat loc-rel-cat loc-own-cat
          ))
(1 (trace defineNoun))
(2 (trace defineNoun defineNounTeaching defineNounLexicographic))
(3 (trace defineNoun defineNounTeaching defineNounLexicographic
          indiv_struct structureOfAll findStructure findProbableStructure
          findPossibleStructure findProperties findProbableProperties
          findClassInclusions findProbableClassInclusions
          findPossibleClassInclusions
          findPossibleActions findActions findProbableActions
          findPossibleProperties
          findOwners findSynonyms findPossibleSynonyms findSpatial))
(4 (trace defineNoun defineNounTeaching defineNounLexicographic
          structureOfAll findStructure findSpatial
          findProbableStructure struct-rule struct-presume
          findPossibleStructure
          struct-indiv findClassInclusions findPossibleClassInclusions
          class-indiv findProbableClassInclusions class-sub-sup
          class-rule findActions findProbableActions findPossibleActions
          act-object-rule act-object-&-rule act-object-presum-rule
          act-object-presum-&-rule act-object-noun
          obj-act-indiv obj-act-presume-&-rule obj-act-presum-rule
          obj-act-&-rule obj-act-rule indiv_struct
          findProperties findProbableProperties findPossibleProperties
          prop-rule prop-presume prop-indiv findOwners
          owner-rel owner-poss rel-for-owner
          syn-sub-sup syn-syn findPossibleSynonyms
          findSynonyms eliminateDissimilarClasses
          similarSuperclassesp noAntonymsp
          antonymp eliminateDissimilarStructure similarStructurep
          eliminateDissimilarOwners
          similarOwnersp removeElement findNamedIndividuals named-indiv
          findAgents agent-object action-object prop-relation-1
          prop-& prop-&-relation-1 prop-&-relation-1-presume
          prop-relation-1-presume prop-&-presume prop-relation-2
          prop-&-relation-2 prop-relation-2-presume
          prop-&-relation-2-presume prop-indiv prop-relation-1-indiv
```

```
                prop-relation-2-indiv obj-rel-1 obj-&-rel-1 obj-rel-2
                obj-&-rel-2 obj-rel-1-presume obj-&-rel-1-presume
                obj-rel-2-presume obj-&-rel-2-presume obj-rel-1-indiv
                obj-rel-2-indiv loc-prop loc-cls loc-str loc-act-obj
                loc-rel loc-own loc-prop-cat loc-cls-cat loc-str-cat
                loc-act-obj-cat loc-rel-cat loc-own-cat
                ))
    ))

;;;-------------------------------------------------------------------------
;;;
;;;     function: prettyPrintDef
;;;                                                        created: stn 2002
;;;-------------------------------------------------------------------------
(defun prettyPrintDef (definition)
  "Prints human readable version of the definition generated by the
    algorithm to standard output."
  (format t "~& Definition of ~A: " (ndefn-noun definition))

  (if (not (null (ndefn-classInclusions definition)))
      (format t "~& Class Inclusions: ~{~A, ~}"
              ;; call lexicalize on each element of the list of
              ;;  class inclusions and then print each of them,
              ;;  separated by commas
              (report (ndefn-classInclusions definition))))
  (if (not (null (ndefn-probableClassInclusions definition)))
      (format t "~& Probable Class Inclusions: ~{~A, ~}"
              (report (ndefn-probableClassInclusions definition))))
  (if (not (null (ndefn-possibleClassInclusions definition)))
      (format t "~& Possible Class Inclusions: ~{~A, ~}"
              (report (ndefn-possibleClassInclusions definition))))
  (if (not (null (ndefn-structuralElements definition)))
      (format t "~& Structure: ~{~A, ~}"
              (report (ndefn-structuralElements definition))))
  (if (not (null (ndefn-probableStructuralElements definition)))
      (format t "~& Probable Structure: ~{~A, ~}"
              (report (ndefn-probableStructuralElements definition))))
  (if (not (null (ndefn-possibleStructuralElements definition)))
      (format t "~& Possible Structure: ~{~A, ~}"
              (report (ndefn-possibleStructuralElements definition))))
;;; If functions are put back into the algorithm, they will go here
  (if (not (null (ndefn-actions definition)))
      (format t "~& Actions: ~{~A, ~}"
              (report (ndefn-actions definition))))
  (if (not (null (ndefn-probableActions definition)))
      (format t "~& Probable Actions: ~{~A, ~}"
              (report (ndefn-probableActions definition))))
  (if (not (null (ndefn-possibleActions definition)))
      (format t "~& Possible Actions: ~{~A, ~}"
              (report (ndefn-possibleActions definition))))
  (if (not (null (ndefn-agents definition)))
      (format t "~& Actions performed on a ~A: ~{~A, ~}"
              (ndefn-noun definition)
              (report (ndefn-agents definition))))
  (if (not (null (ndefn-properties definition)))
      (format t "~& Properties: ~{~A, ~}"
              (report (ndefn-properties definition))))
  (if (not (null (ndefn-probableProperties definition)))
```

```
      (format t "~& Probable Properties: ~{~A, ~}"
              (report (ndefn-probableProperties definition))))
  (if (not (null (ndefn-possibleProperties definition)))
      (format t "~& Possible Properties: ~{~A, ~}"
              (report (ndefn-possibleProperties definition))))
  (if (not (null (ndefn-spatial definition)))
      (format t "~& ~A is a place where: ~{~A, ~}"
              (ndefn-noun definition)
              (report (ndefn-spatial definition))))
  (if (not (null (ndefn-owners definition)))
      (format t "~& Possessive: ~{~A, ~}"
              (report (ndefn-owners definition))))
  (if (not (null (ndefn-synonyms definition)))
      (format t "~& Synonyms: ~{~A, ~}"
              (report (ndefn-synonyms definition))))
  (if (not (null (ndefn-possibleSynonyms definition)))
      (format t "~& Possibly Similar Items: ~{~A, ~}"
              (report (ndefn-possibleSynonyms definition))))
  (if (not (null (ndefn-namedIndividuals definition)))
      (format t "~& Named Individuals: ~{~A, ~}"
              (report (ndefn-namedIndividuals definition))))
  )

;;;-------------------------------------------------------------------------
;;;
;;;     function: report
;;;                                                        created: stn 2003
;;;-------------------------------------------------------------------------
(defun report (nodes)
  "Returns a list of the human language representations of the input list
    of nodes with any duplicates removed."
  (cond ((null nodes) nil)
        (t (union (lexicalize (first nodes))
                  (report (rest nodes)) :test #'string=))))

;;;-------------------------------------------------------------------------
;;;
;;;     function: lexicalize
;;;                                                        created: stn 2002
;;;-------------------------------------------------------------------------
(defun lexicalize (nodes)
  "Finds and returns the human language representation of the sneps
    nodes listed in 'nodes' if one exists. If no human language
    representation can be found, the node itself is returned."
  (let (humanRep)
    (cond
     ;; if the list is empty return the empty string
     ((null nodes) nil)
     ;; if we have a list of lists process each sublist individually
     ((and (listp nodes) (listp (first nodes)))
           (append (lexicalize (first nodes)) (lexicalize (rest nodes))))
     ;; if we have a list consisting of two nodes, process them and
     ;;   concatenate the result
     ((and (listp nodes) (eql (length nodes) 2)
           (not (listp (first nodes))) (not (listp (second nodes))))
      (list (concatenate 'string (first (lexicalize (first nodes))) " "
                         (first (lexicalize (second nodes))))))
```

```
      ;; look for lex arcs coming from the node
      ((setf humanRep #3! ((find lex- ~nodes)))
       (list (nodes2string humanRep)))
      ;; look for mod/head arcs coming from the node
      ((setf humanRep (append #3! ((find (compose mod- lex-) ~nodes))
                              #3! ((find (compose head- lex-) ~nodes))))
       (list (nodes2string humanRep)))
      ;; if the node itself is not named, see if it is a member of a named class
      ((setf humanRep
         (removeAllSuperclasses
          #3! ((find (compose lex- class- ! member) ~nodes))))
       (list (nodes2string humanRep)))
      ;; if the node is part of a skolem function, just use "something"
      ;;  Note: the setf here is unnecessary, but leaving it out was confusing
      ((setf humanRep #3! ((find skf- ~nodes)))
            (list "something"))

      ;; other possible representations would go here

      ;; if we can't find a human language representation, return the
      ;;  name of the sneps node
      (t (list (nodes2string nodes))))))

;;;-------------------------------------------------------------------------
;;;
;;;     function: removeAllSuperclasses
;;;       Due to path-based-inference making class-sub-sup transitive,
;;;       extraneous superclasses were being added to the definition;
;;;       this function removes them.
;;;       -- This function may need to be applied to other areas of lexicalize.
;;;                                                      created: stn 2002
;;;-------------------------------------------------------------------------
(defun removeAllSuperclasses (inList &optional outList)
  "Returns all elements of the input list that do not have any subclass
    in the list."
  ;; if we are done checking the incoming list, check the outgoing list
  (if (null inList) outList
    (let (subs)
      ;; find all subclasses of the first element of inList
      (setf subs #3! ((find (compose lex- subclass- ! superclass lex)
                            ~(first inList))))
      ;; if there are no subclasses in the rest of the input list
      ;;  or in the current output list
      (if (null (intersection subs (append (rest inList) outList)))
          ;; add the element to the output list and process the rest
          ;;   of the input list
          (removeAllSuperclasses (rest inList) (cons (first inList) outList))
        ;; otherwise, omit the element from the output list and
        ;;   process the rest
        (removeAllSuperclasses (rest inList) outList)))))

;;;-------------------------------------------------------------------------
;;;
;;;     function: nodes2string
;;;                                                      created: stn 2002
;;;-------------------------------------------------------------------------
(defun nodes2string (nodes)
  "Converts a list of sneps nodes into a string consisting of the names
```

```
    of the nodes, separated as spaces."
  (cond ((null nodes) "")
        ((not (listp nodes)) (getNodeString nodes))
        ;; this is just a hack to remove extra spaces that were showing up
        ((and (eql (length nodes) 1) (not (listp (first nodes))))
         (getNodeString (first nodes)))
        (t (concatenate 'string (getNodeString (first nodes))
                        " " (nodes2string (rest nodes))))))

;;;-------------------------------------------------------------------------
;;;                          CLASS INCLUSIONS SECTION
;;;-------------------------------------------------------------------------

;;;-------------------------------------------------------------------------
;;;
;;;     function: findClassInclusions
;;;                                                      created: stn 2002
;;;-------------------------------------------------------------------------
(defun findClassInclusions (noun)
  "Find all superclasses of 'noun'.  Find all things Y such that, if X
    is a 'noun' then X is a Y."
  (let (superclasses)
    ;; see if we can infer any class relationships that we don't
    ;;  explicitly know about yet -- after inference we will find this
    ;;  info in subsequent steps
     ;#3! ((deduce superclass (build lex $spr) subclass (build lex ~noun)))
    ;; now extract any relevant info
     (cond
      ;; get superclasses represented using sub-sup case frame,
      ;;  in definition mode return them, in teaching mode
      ;;  continue accumulating information
      ((and (setf superclasses (append superclasses (class-sub-sup noun)))
            *dmode*)
       superclasses)
      ;; superclasses represented using a rule
      ((and (setf superclasses (append superclasses (class-rule noun)))
            *dmode*)
       superclasses)

      ;; if we are in teaching mode, return all the accumulated info
      ;; if we are in definition mode, superclasses must be nil here,
      ;;  so return nil
      (t superclasses)))
   )

;;;-------------------------------------------------------------------------
;;;
;;;     function: findProbableClassInclusions
;;;                                                      created: stn 2002
;;;-------------------------------------------------------------------------
(defun findProbableClassInclusions (noun)
  "Find all superclasses of 'noun' that are marked with the
    'mode presumably' tag."
   (let (superclasses)
     (cond
      ;; SN: I don't know why Ehrlich uses deduce here, this is the
      ;;  only place in the algorithm that it is used -- I am leaving
      ;;  it as it is until I understand it
```

```
      ((and *dmode* #3! ((deduce mode (build lex "presumably")
                                 object (build subclass (build lex ~noun)
                                                        superclass (build lex $maybesuper
)))))
       (setf superclasses (append superclasses (class-sub-sup-presum noun))))
      ;; superclasses represented using a presumable rule
      ((and (setf superclasses (append superclasses (class-rule-presum noun)))
            *dmode*)
       superclasses)

      ;; if we are in teaching mode, return all the accumulated info
      ;; if we are in definition mode, superclasses must be nil here,
      ;;   so return nil
      (t superclasses)))
  )

;;;-------------------------------------------------------------------------
;;;
;;;      function: findPossibleClassInclusions
;;;                                                        created: stn 2002
;;;-------------------------------------------------------------------------
(defun findPossibleClassInclusions (noun classIncls)
  "Find possible superclasses of noun.  If some X is a member of the
    class 'noun' and X is also a member of the class Y, then Y is
    listed as a possible class inclusion for 'noun'."
  ;; eliminate any items that would be duplicates of class inclusions
  ;;  or probable class inclusions from the list of possible class inclusions
   (set-difference (class-indiv noun)
                   ;; eliminate classIncls and the noun itself from the list
                   (append classIncls #3! ((find lex ~noun)))))

;;;-------------------------------------------------------------------------
;;;
;;;      function: class-sub-sup
;;;                                                        created: stn 2002
;;;-------------------------------------------------------------------------
(defun class-sub-sup (noun)
  "Finds superclasses of 'noun' represented using the subclass/superclass
    case frame."
  #3! ((find (compose superclass- ! subclass lex) ~noun)))

;;;-------------------------------------------------------------------------
;;;
;;;      function: class-rule
;;;                                                        created: stn 2002
;;;-------------------------------------------------------------------------
(defun class-rule (noun)
  "Finds superclasses of 'noun' represented using a rule"
  #3! ((find (compose class- cq- ! ant class lex) ~noun
             (compose class- member member- class lex) ~noun)))

;;;-------------------------------------------------------------------------
;;;
;;;      function: class-sub-sup-presum
;;;                                                        created: stn 2002
;;;-------------------------------------------------------------------------
(defun class-sub-sup-presum (noun)
  "Finds things that are presumably superclasses of 'noun'."
```

```
  #3! ((find (compose superclass- subclass lex) ~noun
             (compose superclass- object- ! mode lex) "presumably")))

;;;-------------------------------------------------------------------------
;;;
;;;     function: class-rule-presum
;;;                                                        created: stn 2002
;;;-------------------------------------------------------------------------
(defun class-rule-presum (noun)
  "Finds superclasses of 'noun' represented using a rule"
  #3! ((find (compose class- object- cq- ! ant class lex) ~noun
             (compose class- object- mode lex) "presumably"
             (compose class- member member- class lex) ~noun)))

;;;-------------------------------------------------------------------------
;;;
;;;     function: class-indiv
;;;                                                        created: stn 2002
;;;-------------------------------------------------------------------------
(defun class-indiv (noun)
  "Finds possible level superclasses of a level noun."
    #3! ((find (compose class- ! member member- ! class lex) ~noun)))

;;;-------------------------------------------------------------------------
;;;
;;;     function: classFilter
;;;     input:  a list of superclasses as output by "classes", an empty list,
;;;             and the noun to be defined.
;;;     output: a list of classes not redundant with the rest of the definition
;;;     calls:  mostSpecificSuperclassp, classFilter recursively
;;;-------------------------------------------------------------------------
(defun classFilter (superclasses noun &optional filtered)
  "Removes any superclasses of 'noun' which less specific than other
    superclasses of 'noun' from the list of class inclusions and returns
    the result.  For an example of the heirarchy of class inclusions see
    the documentation for mostSpecificSuperclassp."
    (cond ((null superclasses) filtered)
                                    ;;if first element of input is a list
          ((listp (first superclasses))      ;;add classFilter of car &
           (append filtered                  ;;classFilter of cdr to output
                   (list (classFilter (first superclasses) noun filtered))
                   (classFilter (rest superclasses) noun filtered)))
                                    ;;if car input is an ok atom
                                    ;;add it and classFilter of
                                    ;;cdr to output.
          ((basicLevelp (first superclasses))
           (list (first superclasses)))
          ((mostSpecificSuperclassp (first superclasses) noun)
           (append filtered (list (first superclasses))
                   (classFilter (rest superclasses) noun filtered)))
                        ;;otherwise car input not ok.
                        ;;add classFilter of cdr to output.
          (t (classFilter (rest superclasses) noun filtered))))

;;;-------------------------------------------------------------------------
;;;
;;;     function: basicLevelp
;;;                                                        created: stn 2002
```

```
;;;-------------------------------------------------------------------------
(defun basicLevelp (word)
  "If 'word' is basic level return T else return nil."
  #3! ((deduce member ~word class (build lex "basic ctgy"))))

;;;-------------------------------------------------------------------------
;;;
;;;     function: mostSpecificSuperclassp  (a predicate)
;;;     input:  a noun to be defined and a superclass attributed to <noun>
;;;     returns nil if the class can be deduced from other elements of the
;;;             definition, t otherwise.
;;;-------------------------------------------------------------------------
(defun mostSpecificSuperclassp (class noun)
  "Returns t if there are no classes between 'class' and 'noun' in a
    superclass-subclass relation, nil otherwise.  For example if
    class = vertebrate, noun = cat and vertebrate is a superclass of
    mammal which is a superclass of cat, mostSpecificSuperclassp would
    return nil because mammal is a class between cat and vertebrate."
  (not #3! ((find (compose superclass- ! subclass lex) ~noun
                  (compose subclass- ! superclass) ~class))))


;;;-------------------------------------------------------------------------
;;;                              ACTIONS SECTION
;;;-------------------------------------------------------------------------

;;;-------------------------------------------------------------------------
;;;
;;;     function: act-object-rule
;;;                                                      created: stn 2002
;;;-------------------------------------------------------------------------
(defun act-object-rule (noun)
  "Finds actions performed by all 'noun's and the objects that those
    actions are performed on"
  (let (actions)
    (setf actions
      #3! ((find (compose action- act- ! cq- ! ant ! class lex) ~noun
                 (compose action- act- ! agent member- ! class lex) ~noun)))
    ;; find objects associated with each of the actions
    (mapcar #'(lambda (act) (obj-act-rule noun act)) actions)))

;;;-------------------------------------------------------------------------
;;;
;;;     function: obj-act-rule
;;;                                                      created: stn 2002
;;;-------------------------------------------------------------------------
(defun obj-act-rule (noun action)
  "Finds the objects that 'noun' performs 'action' on."
  (let (objects)
    (setf objects
      #3! ((find (compose object- act- ! cq- ! ant ! class lex) ~noun
                 (compose object- action) ~action)))
    (if (null objects) action
      (mapcar #'(lambda (obj) (list action obj)) objects))))

;;;-------------------------------------------------------------------------
;;;
;;;     function: act-object-&-rule
```

```
;;;                                                          created: stn 2002
;;;-------------------------------------------------------------------------
(defun act-object-&-rule (noun)
  "Finds actions performed by all 'noun's and the objects that those
    actions are performed on"
  (let (actions)
    (setf actions
      #3!((find (compose action- act- ! cq- ! &ant ! class lex) ~noun
                (compose action- act- ! agent member- ! class lex) ~noun)))
    (mapcar #'(lambda (act) (obj-act-&-rule noun act)) actions)))

;;;-------------------------------------------------------------------------
;;;
;;;     function: obj-act-&-rule
;;;                                                          created: stn 2002
;;;-------------------------------------------------------------------------
(defun obj-act-&-rule (noun action)
  "Finds the objects that 'noun' performs 'action' on."
  (let (objects)
    (setf objects
      #3!((find (compose object- act- ! cq- ! &ant ! class lex) ~noun
                (compose object- action) ~action)))
    (if (null objects) action
      (mapcar #'(lambda (obj) (list action obj)) objects))))

;;;-------------------------------------------------------------------------
;;;
;;;     function: act-object-presum-rule
;;;                                                          created: stn 2002
;;;-------------------------------------------------------------------------
(defun act-object-presum-rule (noun)
  "Finds actions that are presumed to be performed by all 'noun's and the
    objects that those actions are presumed to be performed on."
  (let (actions)
    (setf actions
      #3!((find (compose action- act- object- mode lex) "presumably"
                (compose action- act- object- cq- ! ant class lex) ~noun
                (compose action- act- agent member- class lex) ~noun)))
    (mapcar #'(lambda (act) (obj-act-presum-rule noun act)) actions)))

;;;-------------------------------------------------------------------------
;;;
;;;     function: obj-act-presum-rule
;;;                                                          created: stn 2002
;;;-------------------------------------------------------------------------
(defun obj-act-presum-rule (noun action)
  "Finds the objects that 'noun' presumably performs 'action' on."
  (let (objects)
    (setf objects
      #3!((find (compose object- act- object- cq- ! ant class lex) ~noun
                (compose object- action) ~action)))
    (if (null objects) action
      (mapcar #'(lambda (obj) (list action obj)) objects))))

;;;-------------------------------------------------------------------------
;;;
;;;     function: act-object-presum-&-rule
;;;                                                          created: stn 2002
```

```
;;;-------------------------------------------------------------------------
(defun act-object-presum-&-rule (noun)
  "Finds actions that are presumed to be performed by all 'noun's and
    the objects that those actions are presumed to be performed on."
  (let (actions)
    (setf actions
      #3!((find (compose action- act- object- mode lex) "presumably"
                (compose action- act- object- cq- !&ant class lex) ~noun
                (compose action- act- agent member- class lex) ~noun)))
    (mapcar #'(lambda (act) (obj-act-presume-&-rule noun act)) actions)))

;;;-------------------------------------------------------------------------
;;;
;;;     function: obj-act-presum-&-rule
;;;                                                      created: stn 2002
;;;-------------------------------------------------------------------------
(defun obj-act-presume-&-rule (noun action)
  "Finds the objects that 'noun' presumably performs 'action' on."
  (let (objects)
    (setf objects
      #3!((find (compose object- act- object- cq- !&ant class lex) ~noun
                (compose object- action) ~action)))
    (if (null objects) action
      (mapcar #'(lambda (obj) (list action obj)) objects))))

;;;-------------------------------------------------------------------------
;;;
;;;     function: act-object-noun
;;;                                                      created: stn 2002
;;;-------------------------------------------------------------------------
(defun act-object-noun (noun)
    "Finds actions performed by at least one member of the category 'noun' and
      the objects that those actions are performed on."
  (let (actions)
    (setf actions
      #3!((find (compose action- act- ! agent member- ! class lex) ~noun)))
    (mapcar #'(lambda (act) (obj-act-indiv noun act)) actions)))

;;;-------------------------------------------------------------------------
;;;
;;;     function: obj-act-indiv
;;;                                                      created: stn 2002
;;;-------------------------------------------------------------------------
(defun obj-act-indiv (noun action)
  "Finds the objects that 'noun' performs 'action' on."
  (let (objects)
    (setf objects
      #3!((find (compose object- act- ! agent member- ! class lex) ~noun
                (compose object- action) ~action)))
    (if (null objects) action
      (mapcar #'(lambda (obj) (list action obj)) objects))))

;;;-------------------------------------------------------------------------
;;;
;;;     function: findActions
;;;                                                      created: stn 2002
;;;-------------------------------------------------------------------------
(defun findActions (noun)
```

```
  "Find actions (and the objects that those actions are performed on,
    if any) that are performed by all 'noun's."
  (let (results indivNoun)
    ;; get an individual noun so that we can use it in the deduce
    (setf indivNoun (first #3! ((find (compose member- ! class lex) ~noun))))
    ;; see if we can infer any actions that we don't explicitly kow about yet
    (if (not (null indivNoun))
        #3! ((deduce agent ~indivNoun act $someAct)))
    ;; now extract any relevant info
    (cond
     ;;definite rule, or-entail
     ((and (setf results (append results (act-object-rule noun))) *dmode*)
      results)
     ;;definite rule,&-entail
     ((and (setf results (append results (act-object-&-rule noun))) *dmode*)
      results)

     ;; If we are in teaching mode, return all the info we have accumulated.
     ;; If we are in definition mode results = nil so return nil.
     (t results))))

;;;-------------------------------------------------------------------------
;;;
;;;     function: findProbableActions
;;;                                                          created: stn 2002
;;;-------------------------------------------------------------------------
(defun findProbableActions (noun)
  "Find actions (and the objects that those actions are performed on,
    if any) that can be presumed to be performed by all 'noun's."
  (let (results)
    (cond
     ;;"presumably" rule, or-entail, transitive
     ((and (setf results (append results (act-object-presum-rule noun)))
           *dmode*)
      results)
     ;;"presumably" rule,&-entail, transitive
     ((and (setf results (append results (act-object-presum-&-rule noun)))
           *dmode*)
      results)

     ;; If we are in teaching mode, return all the info we have accumulated.
     ;; If we are in definition mode results = nil so return nil.
     (t results))))

;;;-------------------------------------------------------------------------
;;;
;;;     function: findPossibleActions
;;;     input:  a noun to be defined
;;;     output: a list of actions attributed to any object of type <noun>
;;;                                                        modified: mkb 2002
;;;                                                        modified: stn 2002
;;;-------------------------------------------------------------------------
(defun findPossibleActions (noun)
  "Find actions (and the objects that those actions are performed on,
    if any) that are performed by at least one 'noun'."
  (act-object-noun noun))

;;; THE FOLLOWING IN UNNECESSARILY COMPLICATED BECAUSE WE ONLY HAVE ONE
```

```
;;; CHECK TO DO IF MORE CHECKS ARE ADDED IN THE FUTURE THIS VERSION OF
;;; FINDPOSSIBLEACTIONS SHOULD BE UNCOMMENTED AND USED. -- IF WE BECOME
;;; REASONABLY SURE THAT NO MORE CHECKS WILL BE ADDED THEN WE SHOULD JUST
;;; RENAME ACT-OBJECT-NOUN TO FINDPOSSIBLEACTIONS

;;;  (let (results)
;;;    (cond
;;;     ;; find actions performed by at least one member of the class 'noun'
;;;     ((and (setf results (append results (act-object-noun noun)))
;;;           *dmode*)
;;;      results)
;;;
;;;     ;; if we are in teaching mode, return all the information that we
;;;     ;; accumulated above,
;;;     ;; otherwise, return results (= nil).
;;;     (t results))))

;;;-----------------------------------------------------------------------
;;;
;;;     function: act_filter
;;;     input:  a list of actions as output by "acts", an empty list, and the
;;;             noun to be defined.
;;;     output: a list of actions not redundant with the rest of the definition
;;;     calls:  non_redundant_act, act_filter recursively
;;;-----------------------------------------------------------------------
(defun act_filter (act-list noun &optional filtered)
  (cond
   ;; if we are done filtering, return the filtered list
   ((null act-list)  filtered)
   ;; if the first element is a list, filter recursively
   ((listp (first act-list))
    (append filtered
            (list (act_filter (first act-list) noun filtered))
            (act_filter (rest act-list) noun filtered)))
   ;; if first element is not redundant add it to filtered and filter rest
   ((non_redundant_act (first act-list) noun)
    (append filtered (list (first act-list))
            (act_filter (rest act-list) noun filtered)))
   ;; if first element is redundant, just filter rest of list
   (t (act_filter (rest act-list) noun filtered))))
;;;-----------------------------------------------------------------------
;;;
;;;     function: non_redundant_act  (a predicate)
;;;     input:  a noun to be defined and an act attributed to <noun>
;;;     returns nil if the act can be deduced from other elements of the
;;;             definition, t otherwise.
;;;-----------------------------------------------------------------------
(defun non_redundant_act (act noun)
  (cond
   ;; definite case
   (#3! ((find (compose superclass- ! subclass lex) ~noun
               (compose class- ant- ! cq act action) ~act)) nil)
   ;; presumably
   (#3! ((find (compose superclass- ! subclass lex) ~noun
               (compose class- ant- ! cq object act action) ~act)) nil)
   (t t)))

;;;-----------------------------------------------------------------------
```

```
;;;                                STRUCTURE SECTION
;;;-------------------------------------------------------------------------


;;;-------------------------------------------------------------------------
;;;
;;;      function: findStructure
;;;
;;;        Used for finding structure of a noun.
;;;                                                      modified: mkb 2002
;;;                                                      modified: stn 2002
;;;-------------------------------------------------------------------------
(defun findStructure (noun)
  "Attempts to find structural features common to all members of the
    class noun."
  (struct-rule noun))

;;;-------------------------------------------------------------------------
;;;
;;;      function: findProbableStructure
;;;                                                       created: stn 2002
;;;-------------------------------------------------------------------------
(defun findProbableStructure (noun)
  "Attempts to find structural features that are presumably part of all
    'noun's."
  (struct-presume noun))

;;;-------------------------------------------------------------------------
;;;
;;;      function: findPossibleStructure
;;;                                                       created: stn 2002
;;;-------------------------------------------------------------------------
(defun findPossibleStructure (noun superclasses)
  "Find things that are part of some (but not necessarily all) members of
    the class 'noun', and are also not part of all members of a superclass
    of 'noun'.  For example: if the knowledge base says 'Dogs are Mammals.
    Rover is a Dog. Rover has nose. All Mammals have fur.' Then 'nose'
    would be returned but 'fur' would not be returned."
    (set-difference (indiv_struct noun) (structureOfAll superclasses)))

;;;-------------------------------------------------------------------------
;;;
;;;      function: struct
;;;                                                       created: stn 2002
;;;-------------------------------------------------------------------------
(defun struct-rule (noun)
  "Find any things that are a part of all members of the category 'noun'."
  #3! ((find (compose part- whole member- class lex) ~noun
             (compose part- cq- ! ant class lex) ~noun
             (compose part- whole forall- ! ant class lex)
             ~noun)))

;;;-------------------------------------------------------------------------
;;;
;;;      function: struct-presume
;;;                                                       created: stn 2002
;;;-------------------------------------------------------------------------
(defun struct-presume (noun)
  "Find any things that are presumably a part of all members of the
```

```
    category 'noun'."
  #3! ((find (compose part- whole member- class lex) ~noun
             (compose part- object- mode lex) "presumably"
             (compose part- whole forall- ! ant class lex)
             ~noun)))

;;;-------------------------------------------------------------------------
;;;
;;;     function: struct-indiv
;;;                                                          created: stn 20
02
;;;-------------------------------------------------------------------------
(defun struct-indiv (noun)
  "Find any things that are part of some individual who is a member of the
     category 'noun'."
  #3! ((find (compose part- ! whole member- ! class lex) ~noun)))

;;;-------------------------------------------------------------------------
;;;
;;;     function: indiv_struct
;;;     input:   a noun to be defined and a list of its superclasses.
;;;     output:  a list of possessions attributed to individuals
;;;              of class <noun>. (See note on "struct")
;;;                                                   modified: mkb 2002
;;;                                                   modified: stn 2002
;;;-------------------------------------------------------------------------
(defun indiv_struct (noun)
  "Find things that are part of some (but not necessarily all) members
    of the class 'noun'."
  (let (parts)
    (cond ((and (setf parts (append parts (struct-indiv noun))) *dmode*)
           parts)
          ;; if we have gotten to this point there are 2 possible scenarios:
          ;;  1) we have found no parts -- so 'parts' = nil
          ;;  2) we are in teaching mode (*dmode* = nil) and we want to
          ;;     return all the info that we accumulated in the steps above.
          (t parts))))

;;;-------------------------------------------------------------------------
;;;
;;;     function: structureOfAll
;;;                                                   modified: stn 2002
;;;-------------------------------------------------------------------------
(defun structureOfAll (classes)
  "Find the structure of all the classes listed as parameters."
  (if (not (null classes))
      (append (findStructure (first classes)) (structureOfAll (rest classes)))
  nil))

;;;-------------------------------------------------------------------------
;;;                                PROPERTIES SECTION
;;;-------------------------------------------------------------------------

;;;-------------------------------------------------------------------------
;;;
;;;     function: prop-rule
;;;                                                   created: stn 2002
;;;-------------------------------------------------------------------------
```

```
(defun prop-rule (noun)
  "Finds properties belong to all members of the class 'noun', where 'noun'
    is a category."
  #3! ((find (compose property- ! object member- ! class lex) ~noun
             (compose property- ! cq- ! ant ! class lex) ~noun)))

;;;-------------------------------------------------------------------------
;;;
;;;     function: prop-relation-1
;;;                                                        created: stn 2002
;;;-------------------------------------------------------------------------
(defun prop-relation-1 (noun)
  "Finds relationships that all members of the category 'noun' are involved
     in and the other objects that are also in the relation."
  (let (relations)
    (setf relations
      #3! ((find (compose rel- ! object1 member- ! class lex) ~noun
                 (compose rel- ! cq- ! ant ! class lex) ~noun)))
    ;; find the objects associated with each of the relations
    (mapcar #'(lambda (rel) (obj-rel-1 noun rel)) relations)))

;;;-------------------------------------------------------------------------
;;;
;;;     function: obj-rel-1
;;;                                                        created: stn 2002
;;;-------------------------------------------------------------------------
(defun obj-rel-1 (noun relation)
  "Finds the objects that are involved in the specified relation with
    'noun' and returns a list consisting of the relation followed by
    the objects."
  (let (objects)
    (setf objects
      #3! ((find (compose object2- ! object1 member- ! class lex) ~noun
                 (compose object2- ! cq- ! ant ! class lex) ~noun
                 (compose object2- ! rel ) ~relation)))
    (mapcar #'(lambda (obj) (list relation obj)) objects)))

;;;-------------------------------------------------------------------------
;;;
;;;     function: prop-&
;;;                                                        created: stn 2002
;;;-------------------------------------------------------------------------
(defun prop-& (noun)
  "Finds properties belong to all members of the class 'noun'."
  #3! ((find (compose property- ! object member- ! class lex) ~noun
             (compose property- ! cq- ! &ant ! class lex) ~noun)))

;;;-------------------------------------------------------------------------
;;;
;;;     function: prop-&-relation-1
;;;                                                        created: stn 2002
;;;-------------------------------------------------------------------------
(defun prop-&-relation-1 (noun)
  "Finds relationships that all members of the category 'noun' are involved
     in and the other objects that are also in the relation."
  (let (relations)
    (setf relations
      #3! ((find (compose rel- ! object1 member- ! class lex) ~noun
```

```
                (compose rel- ! cq- ! &ant ! class lex) ~noun)))
    ;; find the objects associated with each of the relations
    (mapcar #'(lambda (rel) (obj-&-rel-1 noun rel)) relations)))

;;;-------------------------------------------------------------------------
;;;
;;;     function: obj-&-rel-1
;;;                                                      created: stn 2002
;;;-------------------------------------------------------------------------
(defun obj-&-rel-1 (noun relation)
  "Finds the objects that are involved in the specified relation with 'noun'
    and returns a list consisting of the relation followed by the objects."
  (let (objects)
    (setf objects
      #3!((find (compose object2- ! object1 member- ! class lex) ~noun
                (compose object2- ! cq- ! &ant ! class lex) ~noun
                (compose object2- ! rel ) ~relation)))
    (mapcar #'(lambda (obj) (list relation obj)) objects)))

;;;-------------------------------------------------------------------------
;;;
;;;     function: prop-relation-2
;;;                                                      created: stn 2002
;;;-------------------------------------------------------------------------
(defun prop-relation-2 (noun)
  "Finds relationships that 'noun's are involved in (where the noun is
    the object2) and the other objects that are also in the relation."
  (let (relations)
    (setf relations
      #3!((find (compose rel- ! object2 lex) ~noun
                (compose rel- ! object1 forall- ! cq ! object2 lex) ~noun)))
    ;; find the objects associated with each of the relations
    (mapcar #'(lambda (rel) (obj-rel-2 noun rel)) relations)))

;;;-------------------------------------------------------------------------
;;;
;;;     function: obj-rel-2
;;;                                                      created: stn 2002
;;;-------------------------------------------------------------------------
(defun obj-rel-2 (noun relation)
  "Finds the objects that are involved in the specified relation with
    'noun' and returns a list consisting of the relation followed by the
    objects."
  (let (objects)
    (setf objects
      #3!((find (compose class- ! member object1- ! object2 lex) ~noun
                (compose class- ! ant- ! cq ! object2 lex) ~noun
                (compose class- ! member object1- ! rel ) ~relation)))
    (mapcar #'(lambda (obj) (list obj relation)) objects)))

;;;-------------------------------------------------------------------------
;;;
;;;     function: prop-&-relation-2
;;;                                                      created: stn 2002
;;;-------------------------------------------------------------------------
(defun prop-&-relation-2 (noun)
  "Finds relationships that 'noun's are involved in (where the noun is
    the object2) and the other objects that are also in the relation."
```

```
  (let (relations)
    (setf relations
      #3!((find (compose rel- ! object2 lex) ~noun
                (compose rel- ! object1 forall- ! cq ! object2 lex) ~noun)))
    ;; find the objects associated with each of the relations
    (mapcar #'(lambda (rel) (obj-&-rel-2 noun rel)) relations)))

;;;-------------------------------------------------------------------------
;;;
;;;     function: obj-&-rel-2
;;;                                                      created: stn 2002
;;;-------------------------------------------------------------------------
(defun obj-&-rel-2 (noun relation)
  "Finds the objects that are involved in the specified relation with
    'noun' and returns a list consisting of the relation followed by
    the objects."
  (let (objects)
    (setf objects
      #3!((find (compose class- ! member object1- ! object2 lex) ~noun
                (compose class- ! &ant- ! cq ! object2 lex) ~noun
                (compose class- ! member object1- ! rel ) ~relation)))
    (mapcar #'(lambda (obj) (list obj relation)) objects)))

;;;-------------------------------------------------------------------------
;;;
;;;     function: prop-relation-1-presume
;;;                                                      created: stn 2002
;;;-------------------------------------------------------------------------
(defun prop-relation-1-presume (noun)
  "Finds relationships that all members of the category 'noun' are presumably
     involved in and the other objects that are also in the relation."
  (let (relations)
    (setf relations
      #3!((find (compose rel- ! object1 member- ! class lex) ~noun
                (compose rel- ! object- cq- ! ant ! class lex) ~noun
                (compose rel- ! object- mode lex) "presumably")))
    ;; find the objects associated with each of the relations
    (mapcar #'(lambda (rel) (obj-rel-1-oresume noun rel)) relations)))

;;;-------------------------------------------------------------------------
;;;
;;;     function: obj-rel-1-presume
;;;                                                      created: stn 2002
;;;-------------------------------------------------------------------------
(defun obj-rel-1-presume (noun relation)
  "Finds the objects that are involved in the specified relation with
    'noun' and returns a list consisting of the relation followed by
    the objects."
  (let (objects)
    (setf objects
      #3!((find (compose object2- ! object1 member- ! class lex) ~noun
                (compose object2- ! object- cq- ! ant ! class lex) ~noun
                (compose object2- ! object- mode lex) "presumably"
                (compose object2- ! rel ) ~relation)))
    (mapcar #'(lambda (obj) (list relation obj)) objects)))

;;;-------------------------------------------------------------------------
;;;
```

```
;;;      function: prop-presume
;;;                                                        created: stn 2002
;;;-------------------------------------------------------------------------
(defun prop-presume (noun)
  "Finds properties that presumably belong to all members of the class 'noun'."
  #3! ((find (compose property- object member- ! class lex) ~noun
             (compose property- object- mode lex) "presumably"
             (compose property- object- cq- ! ant class lex) ~noun)))

;;;-------------------------------------------------------------------------
;;;
;;;      function: prop-&-relation-1-presume
;;;                                                        created: stn 2002
;;;-------------------------------------------------------------------------
(defun prop-&-relation-1-presume (noun)
  "Finds relationships that all members of the category 'noun' are presumably
     involved in and the other objects that are also in the relation."
  (let (relations)
    (setf relations
      #3! ((find (compose rel- ! object1 member- ! class lex) ~noun
                 (compose rel- ! object- cq- ! &ant ! class lex) ~noun
                 (compose rel- ! object- mode lex) "presumably")))
    ;; find the objects associated with each of the relations
    (mapcar #'(lambda (rel) (obj-&-rel-1-presume noun rel)) relations)))

;;;-------------------------------------------------------------------------
;;;
;;;      function: obj-&-rel-1-presume
;;;                                                        created: stn 2002
;;;-------------------------------------------------------------------------
(defun obj-&-rel-1-presume (noun relation)
  "Finds the objects that are involved in the specified relation with
    'noun' and returns a list consisting of the relation followed by
    the objects."
  (let (objects)
    (setf objects
      #3! ((find (compose object2- ! object1 member- ! class lex) ~noun
                 (compose object2- ! object- cq- ! &ant ! class lex) ~noun
                 (compose object2- ! object- mode lex) "presumably"
                 (compose object2- ! rel ) ~relation)))
    (mapcar #'(lambda (obj) (list relation obj)) objects)))

;;;-------------------------------------------------------------------------
;;;
;;;      function: prop-&-presume
;;;                                                        created: stn 2002
;;;-------------------------------------------------------------------------
(defun prop-&-presume (noun)
  "Finds properties that presumably belong to all members of the class 'noun'."
  #3! ((find (compose property- ! object member- ! class lex) ~noun
             (compose property- ! object- mode lex) "presumably"
             (compose property- ! object- cq- ! &ant ! class lex) ~noun)))

;;;-------------------------------------------------------------------------
;;;
;;;      function: prop-relation-2-presume
;;;                                                        created: stn 2002
;;;-------------------------------------------------------------------------
```

```
(defun prop-relation-2-presume (noun)
  "Finds relationships that 'noun's are involved in (where the noun
    is the object2) and the other objects that are also in the relation."
  (let (relations)
    (setf relations
      #3!((find (compose rel- ! object2 lex) ~noun
                (compose rel- ! object1 forall- ! cq object ! object2 lex) ~nou
n
                (compose rel- ! object1 forall- ! cq mode lex) "presumably")))
    ;; find the objects associated with each of the relations
    (mapcar #'(lambda (rel) (obj-rel-2-presume noun rel)) relations)))

;;;-------------------------------------------------------------------------
;;;
;;;      function: obj-rel-2-presume
;;;                                                      created: stn 2002
;;;-------------------------------------------------------------------------
(defun obj-rel-2-presume (noun relation)
  "Finds the objects that are involved in the specified relation with 'noun'
    and returns a list consisting of the relation followed by the objects."
  (let (objects)
    (setf objects
      #3!((find (compose class- ! member object1- ! object2 lex) ~noun
                (compose class- ! ant- ! cq object ! object2 lex) ~noun
                (compose class- ! ant- ! cq mode lex) "presumably"
                (compose class- ! member object1- ! rel ) ~relation)))
    (mapcar #'(lambda (obj) (list obj relation)) objects)))

;;;-------------------------------------------------------------------------
;;;
;;;      function: prop-&-relation-2-presume
;;;                                                      created: stn 2002
;;;-------------------------------------------------------------------------
(defun prop-&-relation-2-presume (noun)
  "Finds relationships that 'noun's are involved in (where the noun is the
    object2) and the other objects that are also in the relation."
  (let (relations)
    (setf relations
      #3!((find (compose rel- ! object2 lex) ~noun
                (compose rel- ! object1 forall- ! cq object ! object2 lex) ~nou
n
                (compose rel- ! object1 forall- ! cq mode) "presumably")))
    ;; find the objects associated with each of the relations
    (mapcar #'(lambda (rel) (obj-&-rel-2-presume noun rel)) relations)))

;;;-------------------------------------------------------------------------
;;;
;;;      function: obj-&-rel-2-presume
;;;                                                      created: stn 2002
;;;-------------------------------------------------------------------------
(defun obj-&-rel-2-presume (noun relation)
  "Finds the objects that are involved in the specified relation with 'noun'
    and returns a list consisting of the relation followed by the objects."
  (let (objects)
    (setf objects
      #3!((find (compose class- ! member object1- ! object2 lex) ~noun
                (compose class- ! &ant- ! cq object ! object2 lex) ~noun
                (compose class- ! &ant- ! cq mode lex) "presumably"
```

```
                (compose class- ! member object1- ! rel ) ~relation)))
    (mapcar #'(lambda (obj) (list obj relation)) objects)))

;;;-------------------------------------------------------------------------
;;;
;;;      function: prop-indiv
;;;                                                        created: stn 2002
;;;-------------------------------------------------------------------------
(defun prop-indiv (noun)
  "Finds properties that presumably belong to all members of the class 'noun'."
  #3! ((find (compose property- ! object member- ! class lex) ~noun)))

;;;-------------------------------------------------------------------------
;;;
;;;      function: prop-relation-1-indiv
;;;                                                        created: stn 2002
;;;-------------------------------------------------------------------------
(defun prop-relation-1-indiv (noun)
  "Finds relationships that one or more  members of the category 'noun' are
     involved in and the other objects that are also in the relation."
  (let (relations)
    (setf relations
      #3! ((find (compose rel- ! object1 member- ! class lex) ~noun)))
    ;; find the objects associated with each of the relations
    (mapcar #'(lambda (rel) (obj-rel-1-indiv noun rel)) relations)))

;;;-------------------------------------------------------------------------
;;;
;;;      function: obj-rel-1-indiv
;;;                                                        created: stn 2002
;;;-------------------------------------------------------------------------
(defun obj-rel-1-indiv (noun relation)
  "Finds the objects that are involved in the specified relation with 'noun'
    and returns a list consisting of the relation followed by the objects."
  (let (objects)
    (setf objects #3! ((find (compose object2- ! rel lex lex- rel- ! object1
                                      member- ! class lex) ~noun
                             (compose object2- ! rel) ~relation)))
    (mapcar #'(lambda (obj) (list relation obj)) objects)))

;;;-------------------------------------------------------------------------
;;;
;;;      function: prop-relation-2-indiv
;;;                                                        created: stn 2002
;;;-------------------------------------------------------------------------
(defun prop-relation-2-indiv (noun)
  "Finds relationships that some 'noun's are involved in (where the noun
    is the object2) and the other objects that are also in the relation."
  (let (relations)
    (setf relations #3! ((find (compose rel- ! object2 lex) ~noun)))
    ;; find the objects associated with each of the relations
    (mapcar #'(lambda (rel) (obj-rel-2-indiv noun rel)) relations)))

;;;-------------------------------------------------------------------------
;;;
;;;      function: obj-rel-2-indiv
;;;                                                        created: stn 2002
;;;-------------------------------------------------------------------------
```

```
(defun obj-rel-2-indiv (noun relation)
  "Finds the objects that are involved in the specified relation with 'noun'
    and returns a list consisting of the relation followed by the objects."
  (let (objects)
    (setf objects
      #3! ((find (compose class- ! member object1- ! rel lex lex- rel- !
                          object2 lex) ~noun
                 (compose class- ! member object1- ! rel) ~relation)))
    (mapcar #'(lambda (obj) (list obj relation)) objects)))

;;;-------------------------------------------------------------------------
;;;
;;;     function: findProperties
;;;     input:  a noun to be defined
;;;     output: a list containing any general properties that are known to
;;;             pertain to <noun>s as a class.
;;;                                                      modified: mkb 2002
;;;                                                      modified: stn 2002
;;;-------------------------------------------------------------------------
(defun findProperties (noun)
  "Finds properties that are known to belong to all things which are
    members of the class 'noun'."
  (let (properties)
    (cond
     ;; property of a ctgy.
     ((and (setf properties (append properties (prop-rule noun))) *dmode*)
      properties)
     ;; property of a ctgy, &-rule.
     ((and (setf properties (append properties (prop-& noun))) *dmode*)
      properties)
     ;; relation with 'noun' as object1
     ((and (setf properties (append properties (prop-relation-1 noun)))
           *dmode*)
      properties)
     ;;  relation with 'noun' as object1, &-rule
     ((and (setf properties (append properties (prop-&-relation-1 noun)))
           *dmode*)
      properties)
     ;; relation with noun as object2
     ((and (setf properties (append properties (prop-relation-2 noun)))
           *dmode*)
      properties)
     ;; relation with noun as object2, &-rule
     ((and (setf properties (append properties (prop-&-relation-2 noun)))
           *dmode*)
      properties)

     ;; if we are in teaching mode return the info we have accumulated,
     ;;  if we are in definition mode return nil
     (t properties))))

;;;-------------------------------------------------------------------------
;;;
;;;     function: findProbableProperties
;;;                                                      created: stn 2002
;;;-------------------------------------------------------------------------
(defun findProbableProperties (noun)
  "Finds properties that are presumed to belong to all things which are
```

```
    members of the class 'noun'."
  (let (properties)
    (cond
     ;; presumable property
     ((and (setf properties (append properties (prop-presume noun))) *dmode*)
      properties)
     ;; presumable property &-rule
     ((and (setf properties (append properties (prop-&-presume noun))) *dmode*)
      properties)
     ;; presumable relation, object2 is not in any category
     ((and (setf properties (append properties (prop-relation-1-presume noun)))
           *dmode*)
      properties)
     ;; presumable relation, object2 is not in any category, &-rule
     ((and (setf properties (append properties (prop-&-relation-1-presume noun))
)
           *dmode*)
      properties)
     ;; presumable relation with noun as object2
     ((and (setf properties (append properties (prop-relation-2-presume  noun)))
           *dmode*)
      properties)
     ;; presumable relation with noun as object2
     ((and (setf properties (append properties (prop-&-relation-2-presume noun))
)
           *dmode*)
      properties)

     ;; if we are in teaching mode return the info we have accumulated,
     ;; if we are in definition mode return nil
     (t properties))))

;;;-------------------------------------------------------------------------
;;;
;;;     function: findPossibleProperties
;;;     input:  a noun to be defined
;;;     output: a list of properties attributed to any object of type <noun>
;;;                                                  modified: mkb 2002
;;;                                                  modified: stn 2002
;;;-------------------------------------------------------------------------
(defun findPossibleProperties (noun)
  "Finds properties that belong to at least one thing which is a member
    of the class 'noun'."
  (append
   ;; property belonging to a 'noun'
   (prop-indiv noun)
   ;; relation with 'noun' as object1
   (prop-relation-1-indiv noun)
   ;; relation with noun as object2
   (prop-relation-2-indiv noun)))

;;;-------------------------------------------------------------------------
;;;                            SPATIAL INFORMATION SECTION
;;;-------------------------------------------------------------------------
;;;-------------------------------------------------------------------------
;;;
;;;     function: findSpatial
;;;                                                  created: stn 2002
```

```
;;;-------------------------------------------------------------------------
(defun findSpatial (noun)
  "If 'noun' is a location, find things that can occur in the location
    or that are true in the location"
  (append (loc-cls noun)
          (loc-cls-cat noun)
          (loc-str noun)
          (loc-str-cat noun)
          (loc-act-obj noun)
          (loc-act-obj-cat noun)
          (loc-prop noun)
          (loc-prop-cat noun)
          (loc-rel noun)
          (loc-rel-cat noun)
          (loc-own noun)
          (loc-own-cat noun)))

;;;-------------------------------------------------------------------------
;;;
;;;     function: loc-cls
;;;                                                      created: stn 2002
;;;-------------------------------------------------------------------------
(defun loc-cls (noun)
  "Find things that are members of a class in the location 'noun'"
  (let (mem cls)
    (setf mem #3! ((find (compose member- ! object- ! location lex) ~noun)))
    (setf cls #3! ((find (compose class- ! object- ! location lex) ~noun)))
    (if (and mem cls)
        (list (append mem cls))
      nil)))

;;;-------------------------------------------------------------------------
;;;
;;;     function: loc-cls
;;;                                                      created: stn 2002
;;;-------------------------------------------------------------------------
(defun loc-cls-cat (noun)
  "Find things that are members of a class in the category of locations 'noun'"
  (let (mem cls)
    (setf mem
      #3! ((find (compose member- ! object- ! location member- ! class lex)
                 ~noun)))
    (setf cls
      #3! ((find (compose class- ! object- ! location member- ! class lex)
                 ~noun)))
    (if (and mem cls)
        (list (append mem cls))
      nil)))

;;;-------------------------------------------------------------------------
;;;
;;;     function: loc-str
;;;                                                      created: stn 2002
;;;-------------------------------------------------------------------------
(defun loc-str (noun)
  "Find things that are parts of a whole in the location 'noun'"
  (let (prt whl)
    (setf prt #3! ((find (compose part- ! object- ! location lex) ~noun)))
```

```
    (setf whl #3!((find (compose whole- ! object- ! location lex) ~noun)))
    (if (and whl prt)
        (list (append whl prt))
      nil)))

;;;-------------------------------------------------------------------------
;;;
;;;     function: loc-str-cat
;;;                                                        created: stn 2002
;;;-------------------------------------------------------------------------
(defun loc-str-cat (noun)
  "Find things that are parts of a whole in the category of locations 'noun'"
  (let (prt whl)
    (setf prt
      #3!((find (compose part- ! object- ! location member- ! class lex)
                ~noun)))
    (setf whl
      #3!((find (compose whole- ! object- ! location member- ! class lex)
                ~noun)))
    (if (and whl prt)
        (list (append whl prt))
      nil)))

;;;-------------------------------------------------------------------------
;;;
;;;     function: loc-act-obj
;;;                                                        created: stn 2002
;;;-------------------------------------------------------------------------
(defun loc-act-obj (noun)
  "Find agents, the actions that they perform in the location 'noun', and the
    objects of the actions."
  (let (ag act obj)
    (setf ag #3!((find (compose agent- object- ! location lex) ~noun)))
    (setf act #3!((find (compose action- act- object- ! location lex) ~noun)))
    (setf obj #3!((find (compose object- act- object- ! location lex) ~noun)))
    (if (and ag act obj)
        (list (append ag act obj))
      nil)))

;;;-------------------------------------------------------------------------
;;;
;;;     function: loc-act-obj-cat
;;;                                                        created: stn 2002
;;;-------------------------------------------------------------------------
(defun loc-act-obj-cat (noun)
  "Find agents, the actions that they perform in the category of locations
    'noun', and the objects of the actions."
  (let (ag act obj)
    (setf ag
      #3!((find (compose agent- object- ! location member- ! class lex) ~noun))
)
    (setf act #3!((find (compose action- act- object- ! location
                                 member- ! class lex) ~noun)))
    (setf obj #3!((find (compose object- act- object- ! location
                                 member- ! class lex) ~noun)))
    (if (and ag act obj)
        (list (append ag act obj))
      nil)))
```

```
;;;-------------------------------------------------------------------------
;;;
;;;     function: loc-prop
;;;                                                          created: stn 2002
;;;-------------------------------------------------------------------------
(defun loc-prop (noun)
  "Find objects and their properties which occur in the location 'noun'."
  (let (obj prop)
    (setf obj #3!((find (compose object- ! object- ! location lex) ~noun)))
    (setf prop #3!((find (compose property- ! object- ! location lex) ~noun)))
    (if (and obj prop)
        (list (append obj prop))
      nil)))

;;;-------------------------------------------------------------------------
;;;
;;;     function: loc-prop-cat
;;;                                                          created: stn 2002
;;;-------------------------------------------------------------------------
(defun loc-prop-cat (noun)
  "Find objects and their properties which occur in the category
    of locations 'noun'."
  (let (obj prop)
    (setf obj #3!((find (compose object- ! object- ! location
                                 member- ! class lex) ~noun)))
    (setf prop #3!((find (compose property- ! object- ! location
                                  member- ! class lex) ~noun)))
    (if (and obj prop)
        (list (append obj prop))
      nil)))

;;;-------------------------------------------------------------------------
;;;
;;;     function: loc-rel
;;;                                                          created: stn 2002
;;;-------------------------------------------------------------------------
(defun loc-rel (noun)
  "Find objects that have some relationship in the location 'noun'."
  (let (obj1 relation obj2)
    (setf obj1 #3!((find (compose object1- object- ! location lex) ~noun)))
    (setf obj2 #3!((find (compose object2- object- ! location lex) ~noun)))
    (setf relation #3!((find (compose rel- object- ! location lex) ~noun)))
    (if (and obj1 relation obj2)
        (list (append obj1 relation obj2))
      nil)))

;;;-------------------------------------------------------------------------
;;;
;;;     function: loc-rel-cat
;;;                                                          created: stn 2002
;;;-------------------------------------------------------------------------
(defun loc-rel-cat (noun)
  "Find objects that have some relationship in the category of locations
    'noun'."
  (let (obj1 relation obj2)
    (setf obj1 #3!((find (compose object1- object- ! location
                                 member- ! class lex) ~noun)))
```

```
    (setf obj2 #3!((find (compose object2- object- ! location
                                 member- ! class lex) ~noun)))
    (setf relation #3!((find (compose rel- object- ! location
                                      member- ! class lex) ~noun)))
    (if (and obj1 relation obj2)
        (list (append obj1 relation obj2))
      nil)))

;;;-------------------------------------------------------------------------
;;;
;;;     function: loc-own
;;;                                                          created: stn 2002
;;;-------------------------------------------------------------------------
(defun loc-own (noun)
  "Find owners and the things they own in the location 'noun'."
  (let (owner prpty)
    (setf owner
      #3!((find (compose possessor- ! object- ! location lex) ~noun)))
    ;; I assume that the nature of the property owned by a possesor is
    ;;  best described by the 'rel' arc in the object-rel-possessor
    ;;  case frame (e.g. given object-pyewacket-rel-cat-possessor-person
    ;;  it is better to say "a person owns a cat" than "a person owns
    ;;  pyewacket"
    (setf prpty #3!((find (compose rel- ! object- ! location lex) ~noun)))
    (if (and owner prpty)
        (list (append owner prpty))
      nil)))

;;;-------------------------------------------------------------------------
;;;
;;;     function: loc-own-cat
;;;                                                          created: stn 2002
;;;-------------------------------------------------------------------------
(defun loc-own-cat (noun)
  "Find owners and the things they own in the category of locations 'noun'."
  (let (owner prpty)
    (setf owner #3!((find (compose possessor- ! object- ! location
                                 member- ! class lex) ~noun)))
    (setf prpty #3!((find (compose rel- ! object- ! location
                                 member- ! class lex) ~noun)))
    (if (and owner prpty)
        (list (append owner prpty))
      nil)))

;;;-------------------------------------------------------------------------
;;;                              OWNERSHIP SECTION
;;;-------------------------------------------------------------------------
;;;-------------------------------------------------------------------------
;;;
;;;     function: owner-rel
;;;                                                          created: stn 2002
;;;-------------------------------------------------------------------------
(defun owner-rel (noun)
  "Finds things which own a 'noun', where the noun is specified
    by a 'rel' arc."
  #3!((find (compose class- ! member possessor- ! rel lex) ~noun)))

;;;-------------------------------------------------------------------------
```

```
;;;
;;;     function: owner-poss
;;;                                                          created: stn 2002
;;;---------------------------------------------------------------------------
(defun owner-poss (noun)
  "Finds things which own a 'noun', where some member of the class
    'noun' is the object and the relation between the owner and 'noun'."
  (let (owners rel)
    (setf owners
      #3!((find (compose class- ! member
                         possessor- ! object member- ! class lex) ~noun)))
    ;; find relations associated with each of the owners
    (mapcar #'(lambda (own) (rel-for-owner noun own)) owners)))

;;;---------------------------------------------------------------------------
;;;
;;;     function: rel-for-owner
;;;                                                          created: stn 2002
;;;---------------------------------------------------------------------------
(defun rel-for-owner (noun owner)
  "Finds relations where 'owner' is a possessor of a 'noun'."
  (let (rel)
    ;; find relations
    (setf rel
      #3!((find (compose rel- ! object member- ! class lex) ~noun
                (compose rel- ! possessor member- ! class) ~owner)))
    ;; eliminate the noun itself from the list of relations
    (setf rel (set-difference rel #3!((find lex ~noun)) ))
    ;; if there were any relations other than the noun itself, join them
    ;;  with the owner and return that list, otherwise return nil
    (if (not (null rel))
        (cons owner rel)
      nil)))

;;;---------------------------------------------------------------------------
;;;
;;;     function: findOwners
;;;     input:  a noun to be defined
;;;     output: a list of those things which possess any object of
;;;              type <noun>
;;;
;;;---------------------------------------------------------------------------
(defun findOwners (noun)
  "Find things that can own a 'noun'."
  ;; find owners & get rid of any stray 'nil's that may have crept
  ;;   into the list
  (set-difference (append (owner-rel noun) (owner-poss noun)) '(nil)))

;;;---------------------------------------------------------------------------
;;;                                INDIVIDUALS SECTION
;;;---------------------------------------------------------------------------

;;;---------------------------------------------------------------------------
;;;
;;;     function: findNamedIndividuals
;;;                                                          created: stn 2002
;;;---------------------------------------------------------------------------
(defun findNamedIndividuals (noun)
```

```
  "Find the proper names of individuals who are members of the class noun."
  ;; find individuals
  (named-indiv noun))

;;;-------------------------------------------------------------------------
;;;
;;;     function: named-indiv
;;;                                                        created: stn 2002
;;;-------------------------------------------------------------------------
(defun named-indiv (noun)
  "Finds individuals with proper names who are members of the basic
    level class noun."
  #3! ((find (compose  proper-name- ! object member- ! class lex) ~noun)))

;;;-------------------------------------------------------------------------
;;;                          AGENTS WHO ACT ON 'NOUN'S SECTION
;;;-------------------------------------------------------------------------

;;;-------------------------------------------------------------------------
;;;
;;;     function: agent-object
;;;                                                        created: stn 2002
;;;-------------------------------------------------------------------------
(defun agent-object (noun)
  "Find agents who perform actions on 'noun's and the actions that they
    perform."
  (let (agents)
    (setf agents #3! ((find (compose agent- ! act object
                                     member- ! class lex) ~noun)))
    ;; now find the actions that each agent performs on 'noun's.
    (mapcar #'(lambda (ag) (action-object noun ag)) agents)))

;;;-------------------------------------------------------------------------
;;;
;;;     function: action-object
;;;                                                        created: stn 2002
;;;-------------------------------------------------------------------------
(defun action-object (noun ag)
  "Find actions performed on 'noun's by 'ag'."
  ;; The act- ! act goes up and down the same arc, we need to do this
  ;;  because we only want to find actions that Cassie believes.
  (let (actions)
    (setf actions #3! ((find (compose action- act- ! act object
                                      member- ! class lex) ~noun
                             (compose action- act- ! agent) ~ag)))
    ;; make a list of lists -- the inner lists are pairs of agents and actions
    ;; that they perform -- the outer list consist off all pairs involving the
    ;; same agent
    (mapcar #'(lambda (act) (list ag act)) actions)))

;;;-------------------------------------------------------------------------
;;;
;;;     function: findAgents
;;;     input : a noun to be defined
;;;     returns : a list of the agent(s) and act(s) for which <noun>
;;;               serves as object in an agent-act-object case frame.
;;;
;;;                                                    modified: mkb 04/2002
```

```
;;;                                                    modified: stn 2002
;;;-------------------------------------------------------------------------
(defun findAgents (noun)
  "Find agents who perform actions on 'noun's and the actions that
    they perform."
  (let (agents)
    (cond
     ((and (setf agents (append agents (agent-object noun))) *dmode*)
      agents)

     ;; if we are in teaching mode return the info we have accumulated,
     ;;  else return nil
     (t agents)
     )))

;;;-------------------------------------------------------------------------
;;;                              SYNONYMS SECTION
;;;-------------------------------------------------------------------------

;;;-------------------------------------------------------------------------
;;;
;;;     function: syn-syn
;;;                                                    created: stn 2002
;;;-------------------------------------------------------------------------
(defun syn-syn (noun)
  "Finds things which are explicitly marked as synonyms of 'noun'."
  #3! ((find (compose synonym- ! synonym lex) ~noun)))

;;;-------------------------------------------------------------------------
;;;
;;;     function: syn-sub-sup
;;;                                                    created: stn 2002
;;;-------------------------------------------------------------------------
(defun syn-sub-sup (superclasses)
  "Finds subclasses of the given list of superclasses."
  #3! ((find (compose lex- subclass- ! superclass) ~superclasses)))

;;;-------------------------------------------------------------------------
;;;
;;;     function: findSynonyms
;;;     input:  a noun to be defined
;;;     output: a list of synonyms and possible synonyms of <noun>
;;;
;;;                                                 written: kae ??/??/92
;;;                                                 modified: kae 05/12/94
;;;                                                 modified: stn 2002
;;;-------------------------------------------------------------------------
(defun findSynonyms (noun)
  "Find words that are specifically marked as synonyms of 'noun'."
    ;; find things that are explicitly labeled as synonyms of 'noun'.
    (removeElement noun (syn-syn noun)))

;;;-------------------------------------------------------------------------
;;;     function:  eliminateClassInclusions
;;;-------------------------------------------------------------------------
(defun eliminateClassInclusions (possibleSynonyms superclasses)
  "Remove anything in the list superclasses from the list possibleSynonyms"
  (set-difference possibleSynonyms (report superclasses)
```

```
                    :test #'compareAsString))

;;;-------------------------------------------------------------------------
;;;     function:  compareAsString
;;;-------------------------------------------------------------------------
(defun compareAsString (a b)
  "Compare a and b as strings.  This is for use specifically with the function
    eliminateClassInclusions."
  (string= (getNodeString a) (string b)))

(defun findPossibleSynonyms (noun structuralElements superclasses
                             owners synonyms)
  "Find words that have definitions which are similar to the definition
    of 'noun'."
  (let (possibleSynonyms)
    ;; find things that are subclasses of the same superclasses as 'noun',
    ;;  e.g. if we are trying to define 'cat', and we know that cats and
    ;;  dogs are both subclasses of mammal, then this will find 'dog'.
    (setf possibleSynonyms (syn-sub-sup superclasses))
    ;; since 'noun' is itself a subclass of its superclass, remove
    ;;  'noun' from the list
    (setf possibleSynonyms (removeElement noun possibleSynonyms))
    ;; superclasses are not synonyms, they are class inclusions, so if
    ;;  any snuck into the list of possible synonyms, get rid of them.
    (setf possibleSynonyms
      (eliminateClassInclusions possibleSynonyms superclasses))
    ;; explicit synonyms are obviously not possible synonyms --
    ;;  they are definite synonyms, so we need to get rid of them
    (setf possibleSynonyms (set-difference possibleSynonyms synonyms))

    ;; THE NEXT TWO FUNCTIONS REALLY IMPLEMENT KE'S THEORY OF WHAT IS
    ;;  A POSSIBLY SIMILAR ITEM AND WHAT IS NOT.

    ;; get rid of any possible synonyms whose superclasses are not
    ;;  sufficiently similar to the superclasses of 'noun'
    (setf possibleSynonyms
      (eliminateDissimilarClasses superclasses possibleSynonyms))
    ;; get rid of any possible synonyms whose structural features are not
    ;;  sufficiently similar to the structural features of 'noun'
    (setf possibleSynonyms
      (eliminateDissimilarStructure structuralElements possibleSynonyms))
    ;; get rid of any possible synonyms whose ownership relations are not
    ;;  sufficiently similar to the ownership relations of 'noun'
    ;;(setf possibleSynonyms
    ;;(eliminateDissimilarOwners owners possibleSynonyms nil))

    ;; return the remaining synonmys
    possibleSynonyms))

;;;-------------------------------------------------------------------------
;;;     function: eliminateDissimilarClasses
;;;     input:    supers, a list of the superclasses of the target noun;
;;;               possibleSynonyms, a list of possible synonyms for the
;;;                 target noun
;;;               verifiedSynonyms, a list of those possible synonmys
;;;                 that survive comparison of superclasses (initially nil)
;;;       returns:  verifiedSynonyms
;;;                                              written: kae ??/??/92
```

```
;;;                                                    modified: kae 05/12/94
;;;                                                    modified: stn 2002
;;;-------------------------------------------------------------------------
(defun eliminateDissimilarClasses (supers possibleSynonyms
                                   &optional verifiedSynonyms)
  "Examine each of the elements of 'possibleSynonyms' and eliminate
    any synonyms whose class inclusions are not sufficiently similar
    to the class inclusions of 'noun'."
  (cond
   ;; if there are no more possibe synonyms, return the list of
   ;;  verified synonyms
   ((null possibleSynonyms) verifiedSynonyms)
   ;; if the superclasses of 'noun' are sufficiently similar to the
   ;;  superclasses of the first possible synonym on the list
   ;;  'possibleSynonyms' then add the possible synonym
   ;;  to the list 'verifiedSynonyms' and check the rest of 'possibleSynonyms'
   ((similarSuperclassesp
     supers (union
             (findClassInclusions (first possibleSynonyms))
             (findProbableClassInclusions (first possibleSynonyms))))
    (eliminateDissimilarClasses supers (rest possibleSynonyms)
                        (cons (first possibleSynonyms) verifiedSynonyms)))
   ;; in this case, the sets of superclasses examined above
   ;;  were not sufficiently similar,so do not add the first element of
   ;;  the list 'possibleSynonyms' to 'verifiedSynoynms'
   ;;   (thereby removing it) and check the rest of 'possibleSynonyms'
   (t (eliminateDissimilarClasses supers (rest possibleSynonyms)
                                  verifiedSynonyms))))

;;;-------------------------------------------------------------------------
;;;
;;;     function: similarSuperclassesp  (a predicate)
;;;     input:  two lists of superclasses, superclassesOfNoun is the
;;;             superclasses of the target noun; superclassesOfSynonym
;;;             is the superclasses of a possible synonym.
;;;     returns t if target and possible synonym belong to similar lists of
;;;             superclasses, nil otherwise.
;;;                                                    modified: stn 2002
;;;-------------------------------------------------------------------------
(defun similarSuperclassesp (superclassesOfNoun superclassesOfSynonym)
  "Return t if the two lists of superclasses are sufficiently similar."
  ;; return true if:
  (and
   ;; the two sets of superclass have at least as many elements in common
   ;;  as they have elements which are different
   (>= (length (intersection superclassesOfNoun superclassesOfSynonym))
       (length (union (set-difference superclassesOfNoun superclassesOfSynonym)
                      (set-difference superclassesOfSynonym superclassesOfNoun))
))
   ;; they share at least two superclasses
   (>= (length (intersection superclassesOfNoun superclassesOfSynonym)) 2)
   ;; none of their superclasses are labeled as antonyms
   (noAntonymsp superclassesOfNoun superclassesOfSynonym)))

;;;-------------------------------------------------------------------------
;;;
;;;     function: noAntonymsp  (a predicate)
;;;     input:  two lists of superclasses, superclassesOfNoun is the
```

```
;;;             superclasses of the target noun; superclassesOfSynonym
;;;             is the superclasses of a possible synonym.
;;;     returns nil if an an element of one list has an antonym in the other,
;;;             t otherwise.
;;;-------------------------------------------------------------------------
(defun noAntonymsp (superclassesOfNoun superclassesOfSynonym)
  "Return t if there are no members of 'superclassesOfNoun' which are
    explicitly labeled as an antonym of any member of 'superclassesOfSynonym',
    nil otherwise."
    (cond ((null superclassesOfNoun) t)
          (t (if (null (antonymp (first superclassesOfNoun)
                                 superclassesOfSynonym))
                 (noAntonymsp (rest superclassesOfNoun)
                              superclassesOfSynonym)))))

;;;-------------------------------------------------------------------------
;;;
;;;    function:  antonymp (a predicate)
;;;
;;;-------------------------------------------------------------------------
(defun antonymp (class superclassesOfSynonym)
  "Return t if there is at least one explicitly labeled antonym of 'class'
    among the list of classes 'superclassesOfSynonym', nil otherwise."
  (intersection #3! ((find (compose antonym- ! antonym ) ~class))
                (removeElement (getNodeName class) superclassesOfSynonym)))

;;;-------------------------------------------------------------------------
;;;     function: eliminateDissimilarStructure
;;;     input:    structuralElements, a list of structural elements
;;;                 of the target noun possibleSynonyms, a list of possible
;;;                 synonyms for the target noun verifiedSynonyms, a list of
;;;                 those possible synonmys that survive
;;;                 comparison of structure (initially nil).
;;;     returns:  verifiedSynonyms
;;;                                     written: kae ??/??/92
;;;                                     modified: kae 05/12/94
;;;                                     modified: stn 2002
;;;-------------------------------------------------------------------------
(defun eliminateDissimilarStructure (structuralElements possibleSynonyms
                                     &optional verifiedSynonyms)
  "Examine each element of 'possibleSynonyms' and remove it from the list
    if its structural elements are not sufficiently similar to
    'structuralElements'."
  (cond
   ;; if there are no more possible synonyms to examine, return the list
   ;;  of verified synonyms
   ((null possibleSynonyms) verifiedSynonyms)
   ;; if the first element of 'possibleSynonyms' has a list of structural
   ;;  elements that is sufficiently similar to the structural elements of
   ;;  'noun' then add it to the list of verified synonyms and check the
   ;;  rest of the list.
   ((similarStructurep structuralElements
                       (append (findStructure (first possibleSynonyms))
                               (findProbableStructure (first possibleSynonyms))))
    (eliminateDissimilarStructure structuralElements (rest possibleSynonyms)
                                  (cons (first possibleSynonyms) verifiedSynonyms)))
   ;; don't put the first element of 'possibleSynonyms' into the
   ;;  list of verified synonyms and keep checking the rest of the list
```

```
    (t (eliminateDissimilarStructure structuralElements (rest possibleSynonyms)
                                     verifiedSynonyms))))

;;;-------------------------------------------------------------------------
;;;
;;;     function: similarStructurep (a predicate)
;;;     input:  two lists of structural elementss, structureOfNoun is the
;;;             structure of the target noun; structureOfSynonym is
;;;             the structure of a possible synonym.
;;;     returns t if target and possible synonym have similar lists of
;;;             structural elements, nil otherwise.
;;;-------------------------------------------------------------------------
(defun similarStructurep (structureOfNoun structureOfSynonym)
  "Return t if there are more elements shared by the two sets of input
    lists than elements that separate them."
   (>= (length (intersection structureOfNoun structureOfSynonym))
       (length (union (set-difference structureOfNoun structureOfSynonym)
                      (set-difference structureOfSynonym structureOfNoun)))))

;;;-------------------------------------------------------------------------
;;;     function: eliminateDissimilarOwners
;;;     input:    noun, the target being defined
;;;               possibleSynonyms, a list of possible synonyms for the
;;;               target noun verifiedSynonyms, a list of those possible
;;;               synonmys that survive comparison of ownership or
;;;               part/whole relation (initially nil).
;;;       returns:  verifiedSynonyms
;;;                                     written: kae ??/??/92
;;;                                     modified: kae 05/12/94
;;;-------------------------------------------------------------------------
(defun eliminateDissimilarOwners (owners possibleSynonyms verifiedSynonyms)
  "Examine each element of 'possibleSynonyms' and remove it from the list
    if its owners are not sufficiently similar to the owners of 'noun'."
  (cond
   ;; if there are no more possible synonyms to examine, return the
   ;;  list of verified synonyms
   ((null possibleSynonyms) verifiedSynonyms)
   ;; if the first element of 'possibleSynonyms' has a list of owners that is
   ;;  sufficiently similar to the owners of 'noun' then add it to the list of
   ;;  verified synonyms and check the rest of the list.
   ((similarOwnersp owners (findOwners (first possibleSynonyms)))
    (eliminateDissimilarOwners owners (rest possibleSynonyms)
                (append (list (first possibleSynonyms)) verifiedSynonyms)))
   ;; don't put the first element of 'possibleSynonyms' into the list
   ;;  of verified synonyms and keep checking the rest of the list
   (t (eliminateDissimilarOwners owners (rest possibleSynonyms)
                                 verifiedSynonyms))))

;;;-------------------------------------------------------------------------
;;;
;;;     function: similarOwnersp  (a predicate)
;;;     input:  two lists of relations, ownersOfNoun is the relations of the
;;;             target noun; ownersOfSynonym is the relations of a
;;;             possible synonym.
;;;     returns t if target and possible synonym have similar relations,
;;;             nil otherwise.
;;;-------------------------------------------------------------------------
(defun similarOwnersp (ownersOfNoun ownersOfSynonym)
```

```
  "Return t if the input lists have more elements in common than elements that
     separate them."
    (>= (length (intersection ownersOfNoun ownersOfSynonym))
        (length (union (set-difference ownersOfNoun ownersOfSynonym)
                       (set-difference ownersOfSynonym ownersOfNoun)))))

;;;-------------------------------------------------------------------------
;;;
;;;      function: removeElement
;;;
;;;-------------------------------------------------------------------------
(defun removeElement (removeMe nodeList &optional weeded)
  "Remove an element whose name is equal to the name of 'removeMe' from
    'nodeList'."
  (cond
   ;; if all the elements have been checked, return the list of elements
   ;;   that passed the check
   ((null nodeList) weeded)
   ;; if the name of the node 'removeMe' is the same as the name of the
   ;;  first node in 'nodeList' then do not add the first element to the
   ;;  list of verified elements (weeded) and check the rest of the nodeList
   ((string-equal (string removeMe) (getNodeString (first nodeList)))
    (removeElement removeMe (rest nodeList) weeded))
   ;; if the name of the node that has a lex pointing to 'removeMe' is the
   ;;  same as the first node in 'nodeList' then do not add the first element
   ;;  to the list of verified elementns and check the rest of nodeList
   ((string-equal (getNodeString (first #3!((find lex ~removeMe))))
                  (getNodeString (first nodeList)))
    (removeElement removeMe (rest nodeList) weeded))
   ;; the name of the node 'removeMe' is not the same as the name of the
   ;;  first node in 'nodeList', so add the first node in 'nodeList' to
   ;;  'weeded'.
   (t (removeElement removeMe (rest nodeList)
                     (cons (first nodeList) weeded)))))

;;;-------------------------------------------------------------------------
;;;      function:  getNodeName
;;;-------------------------------------------------------------------------
(defun getNodeName (node)
    (and (sneps:node-p node)
         (sneps:node-na node)))

;;;-------------------------------------------------------------------------
;;;      function:  getNodeString
;;;-------------------------------------------------------------------------
(defun getNodeString (node)
  (string (getNodeName node)))
```

B Demos

B.1 Brachet

```
Starting image '/util/acl62/composer'
  with no arguments
  in directory '/projects/stn2/CVA/demos/'
  on machine 'localhost'.
```

```
International Allegro CL Enterprise Edition
6.2 [Solaris] (Aug 15, 2002 14:24)
Copyright (C) 1985-2002, Franz Inc., Berkeley, CA, USA.  All Rights Reserved.

This development copy of Allegro CL is licensed to:
   [4549] SUNY/Buffalo, N. Campus

;; Optimization settings: safety 1, space 1, speed 1, debug 2.
;; For a complete description of all compiler switches given the current
;; optimization settings evaluate (explain-compiler-settings).
;;---
;; Current reader case mode: :case-sensitive-lower
cl-user(1): :cd ..
/projects/stn2/CVA/
cl-user(2): :ld defun_nounc.
Error: "defun_nounc." does not exist, cannot load
  [condition type: file-does-not-exist-error]

Restart actions (select using :continue):
 0: retry the load of defun_nounc.
 1: skip loading defun_nounc.
 2: Abort entirely from this process.
[1] cl-user(3): :pop
cl-user(4): :ld /projects/snwiz/bin/sneps
; Loading /projects/snwiz/bin/sneps.lisp
Loading system SNePS...10% 20% 30% 40% 50% 60% 70% 80% 90% 100%
SNePS-2.6 [PL:0a 2002/09/30 22:37:46] loaded.
Type `(sneps)' or `(snepslog)' to get started.
cl-user(5): :ld defun_noun.cl
; Loading /projects/stn2/CVA/defun_noun.cl
cl-user(6): (demo "demos/brachet.demo")
Error: attempt to call `demo' which is an undefined function.
  [condition type: undefined-function]

Restart actions (select using :continue):
 0: Try calling demo again.
 1: Try calling sneps:demo instead.
 2: Return a value instead of calling demo.
 3: Try calling a function other than demo.
 4: Setf the symbol-function of demo and call it again.
 5: Return to Top Level (an "abort" restart).
 6: Abort entirely from this process.
[1] cl-user(7): :pop
cl-user(8): (sneps)

   Welcome to SNePS-2.6 [PL:0a 2002/09/30 22:37:46]

Copyright (C) 1984--2002 by Research Foundation of
State University of New York. SNePS comes with ABSOLUTELY NO WARRANTY!
Type `(copyright)' for detailed copyright information.
Type `(demo)' for a list of example applications.

   2/21/2003 13:53:21

* (demo "demos/brachet.demo")

File /projects/stn2/CVA/demos/brachet.demo is now the source of input.
```

```
 CPU time : 0.02

* ;; Brachet demo - created by K. Ehrlich
;;              - modified by A. Hunt & J. Koplas
;;              - modified by M. Broklawski
;;              - modified by S. Napieralski

;; clear all information out of the network
(resetnet t)

Net reset

 CPU time : 0.02

*
;; turn off singular path inference
^(
--> in-package snip)
#<The snip package>

 CPU time : 0.00

*
;;; redefine function to return nil
;;; so that forward inference will not be limited
^(
--> defun broadcast-one-report (rep)
   (let (anysent)
     (do.chset (ch *OUTGOING-CHANNELS* anysent)
               (when (isopen.ch ch)
                 (setq anysent (or (try-to-send-report rep ch) anysent)))))
   nil)
broadcast-one-report

 CPU time : 0.00

*
;;; return to sneps package
^(
--> in-package sneps)
#<The sneps package>

 CPU time : 0.00

*
^
--> (setf snepsul::*infertrace* nil)
nil
 CPU time : 0.00

*
;;; load background knowledge
(intext "/projects/stn2/CVA/demos/rels")
File /projects/stn2/CVA/demos/rels is now the source of input.

 CPU time : 0.01

*
```

```
(a1 a2 a3 a4 after agent antonym associated before cause class direction
 equiv etime from in indobj instr into lex location manner member mode object
 on onto part place possessor proper-name property rel skf sp-rel stime
 subclass superclass synonym time to whole kn_cat)

 CPU time : 0.25

*

End of file /projects/stn2/CVA/demos/rels

 CPU time : 0.26

* (demo "/projects/stn2/CVA/demos/paths")

File /projects/stn2/CVA/demos/paths is now the source of input.

 CPU time : 0.01

* ;;; Composition of certain paths

;;; Make Before Transitive
(define-path before (compose before (kstar (compose after- ! before))))
before implied by the path (compose before (kstar (compose after- ! before)))
before- implied by the path (compose (kstar (compose before- ! after))
                             before-)

 CPU time : 0.00

*
;;; Make After Transitive
(define-path after (compose after (kstar (compose before- ! after))))
after implied by the path (compose after (kstar (compose before- ! after)))
after- implied by the path (compose (kstar (compose after- ! before)) after-)

 CPU time : 0.00

*
;; If X is a member of the class Y and Y is a subclass of Z then
;;   X is a member of the class Z.
;; Example:  If Fido is a brachet and all brachets are hounds then
;;;  Fido is a hound.
(define-path class (compose class (kstar (compose subclass- ! superclass))))
class implied by the path (compose class
                           (kstar (compose subclass- ! superclass)))
class- implied by the path (compose (kstar (compose superclass- ! subclass))
                            class-)

 CPU time : 0.00

*
;; Make subclass transitive
(define-path subclass (compose subclass (kstar (compose superclass- ! subclass))))
subclass implied by the path (compose subclass
                              (kstar (compose superclass- ! subclass)))
subclass- implied by the path (compose
                               (kstar (compose subclass- ! superclass))
```

```
                                subclass-)

 CPU time : 0.00

*

End of /projects/stn2/CVA/demos/paths demonstration.

 CPU time : 0.01

* (demo "/projects/stn2/CVA/demos/brachet.base")

File /projects/stn2/CVA/demos/brachet.base is now the source of input.

 CPU time : 0.00

* ;;; This is the set of assertions which build the base network which
;;; corresponds to Karen Ehrlich's vocabulary acquisition project.
;;; Commented and made accessible by Alan Hunt and Geoffrey Koplas, '97
;;; The following is the information that needs to get fed into the network
;;; for the Narrative Acquisition demos; namely the background knowledge
;;; on the words _other_ than the one being acquired.

; Animals are physical objects
(describe
(assert subclass (build lex "animal") superclass (build lex "phys obj")
        kn_cat "life"))

(m3! (kn_cat life) (subclass (m1 (lex animal)))
 (superclass (m2 (lex phys obj))))

(m3!)

 CPU time : 0.01

*
; Quadrupeds are vertebrates
(describe
(assert subclass (build lex "quadruped") superclass (build lex "vertebrate")
        kn_cat "life"))

(m6! (kn_cat life) (subclass (m4 (lex quadruped)))
 (superclass (m5 (lex vertebrate))))

(m6!)

 CPU time : 0.01

*
; Ungulates are herbivores
(describe
(assert subclass (build lex "ungulate") superclass (build lex "herbivore")
        kn_cat "life"))

(m9! (kn_cat life) (subclass (m7 (lex ungulate)))
 (superclass (m8 (lex herbivore))))

(m9!)
```

```
 CPU time : 0.02

*
; Mammals are animals
(describe
(assert subclass (build lex "mammal") superclass (build lex "animal")
        kn_cat "life"))

(m11! (kn_cat life) (subclass (m10 (lex mammal)))
 (superclass (m1 (lex animal))))

(m11!)

 CPU time : 0.00

*
; Mammals are vertebrates
(describe
(assert subclass (build lex "mammal") superclass (build lex "vertebrate")
        kn_cat "life"))

(m12! (kn_cat life) (subclass (m10 (lex mammal)))
 (superclass (m5 (lex vertebrate))))

(m12!)

 CPU time : 0.02

*
; Deer are mammals
(describe
(assert subclass (build lex "deer") superclass (build lex "mammal")
        kn_cat "life"))

(m14! (kn_cat life) (subclass (m13 (lex deer)))
 (superclass (m10 (lex mammal))))

(m14!)

 CPU time : 0.01

*
; Deer are quadrupeds
(describe
(assert subclass (build lex "deer") superclass (build lex "quadruped")
        kn_cat "life"))

(m15! (kn_cat life) (subclass (m13 (lex deer)))
 (superclass (m4 (lex quadruped))))

(m15!)

 CPU time : 0.00

*
; Deer are herbivores
(describe
```

```
(assert subclass (build lex "deer") superclass (build lex "herbivore")
        kn_cat "life"))

(m16! (kn_cat life) (subclass (m13 (lex deer)))
 (superclass (m8 (lex herbivore))))

(m16!)

 CPU time : 0.01

*
; Deer are animals
(describe (assert subclass (build lex "deer") superclass (build lex "animal")
                  kn_cat "life"))

(m17! (kn_cat life) (subclass (m13 (lex deer)))
 (superclass (m1 (lex animal))))

(m17!)

 CPU time : 0.02

*
; Deer is a basic level category
(describe
(assert member (build lex "deer")  class (build lex "basic ctgy")
        kn_cat "life"))

(m19! (class (m18 (lex basic ctgy))) (kn_cat life) (member (m13 (lex deer))))

(m19!)

 CPU time : 0.02

*
;"Harts are deer"
(describe
(assert subclass (build lex "hart") superclass (build lex "deer")
        kn_cat "life"))

(m21! (kn_cat life) (subclass (m20 (lex hart))) (superclass (m13 (lex deer))))

(m21!)

 CPU time : 0.00

*

; Horses are quadrupeds
(describe
(assert subclass (build lex "horse") superclass (build lex "quadruped")
        kn_cat "life"))

(m23! (kn_cat life) (subclass (m22 (lex horse)))
 (superclass (m4 (lex quadruped))))

(m23!)
```

```
 CPU time : 0.01

*
; Horses are herbivores
(describe
(assert subclass (build lex "horse") superclass (build lex "herbivore")
        kn_cat "life"))

(m24! (kn_cat life) (subclass (m22 (lex horse)))
 (superclass (m8 (lex herbivore))))

(m24!)

 CPU time : 0.01

*
; Horses are animals
(describe
(assert subclass (build lex "horse") superclass (build lex "animal")
        kn_cat "life"))

(m25! (kn_cat life) (subclass (m22 (lex horse)))
 (superclass (m1 (lex animal))))

(m25!)

 CPU time : 0.00

*
; Horse is a basic level category
(describe
(assert member (build lex "horse")  class (build lex "basic ctgy")
        kn_cat "life"))

(m26! (class (m18 (lex basic ctgy))) (kn_cat life) (member (m22 (lex horse))))

(m26!)

 CPU time : 0.00

*
; Ponies are animals
(describe (assert subclass (build lex "pony") superclass (build lex "animal")
                 kn_cat "life"))

(m28! (kn_cat life) (subclass (m27 (lex pony)))
 (superclass (m1 (lex animal))))

(m28!)

 CPU time : 0.01

*
;"Dogs are mammals"
(describe
(assert subclass (build lex "dog") superclass (build lex "mammal")
        kn_cat "life"))
```

```
(m30! (kn_cat life) (subclass (m29 (lex dog)))
 (superclass (m10 (lex mammal))))

(m30!)

 CPU time : 0.01

*
;"Dogs are quadrupeds"
(describe
(assert subclass (build lex "dog") superclass (build lex "quadruped")
        kn_cat "life"))

(m31! (kn_cat life) (subclass (m29 (lex dog)))
 (superclass (m4 (lex quadruped))))

(m31!)

 CPU time : 0.00

*
;"Dogs are animals"
(describe
(assert subclass (build lex "dog") superclass (build lex "animal")
        kn_cat "life"))

(m32! (kn_cat life) (subclass (m29 (lex dog))) (superclass (m1 (lex animal))))

(m32!)

 CPU time : 0.01

*
; Dog is a basic level category
(describe
 (assert member (build lex "dog")  class (build lex "basic ctgy")
        kn_cat "life"))

(m33! (class (m18 (lex basic ctgy))) (kn_cat life) (member (m29 (lex dog))))

(m33!)

 CPU time : 0.00

*
;"Hounds are dogs"
(describe
(assert subclass (build lex "hound") superclass (build lex "dog")
        kn_cat "life"))

(m35! (kn_cat life) (subclass (m34 (lex hound))) (superclass (m29 (lex dog))))

(m35!)

 CPU time : 0.01

*
;"Something is a member of the class dog"
```

```
(describe
(assert member #rex class (build lex "dog")))

(m36! (class (m29 (lex dog))) (member b1))

(m36!)

 CPU time : 0.02

*
;"The dog is possesed by something"
(describe
(assert object *rex rel (build lex "dog") possessor #rexboss))

(m37! (object b1) (possessor b2) (rel (m29 (lex dog))))

(m37!)

 CPU time : 0.00

*
;"The thing that owns the dog is a person"
(describe
(assert member *rexboss class (build lex "person")))

(m39! (class (m38 (lex person))) (member b2))

(m39!)

 CPU time : 0.01

*
; Kings are persons
(describe
(add subclass (build lex "king") superclass (build lex "person")
       kn_cat "life"))

(m41! (kn_cat life) (subclass (m40 (lex king)))
 (superclass (m38 (lex person))))

(m41!)

 CPU time : 0.00

*
; Wizards are persons.
(describe
(assert subclass (build lex "wizard") superclass (build lex "person")
       kn_cat "life"))

(m43! (kn_cat life) (subclass (m42 (lex wizard)))
 (superclass (m38 (lex person))))

(m43!)

 CPU time : 0.01

*
```

```
; Knights are persons.
(describe
(assert subclass (build lex "knight") superclass (build lex "person")
        kn_cat "life"))

(m45! (kn_cat life) (subclass (m44 (lex knight)))
 (superclass (m38 (lex person))))

(m45!)

 CPU time : 0.00

*
; Person is a basic level category
(describe
(assert member (build lex "person")  class (build lex "basic ctgy")
        kn_cat "life"))

(m46! (class (m18 (lex basic ctgy))) (kn_cat life)
 (member (m38 (lex person))))

(m46!)

 CPU time : 0.00

*
; Something is named 'King Arthur'
(describe
 (assert object #KA proper-name (build lex "King Arthur")
         kn_cat "life"))

(m48! (kn_cat life) (object b3) (proper-name (m47 (lex King Arthur))))

(m48!)

 CPU time : 0.01

*
; King Arthur is a king
(describe
(assert member *KA class (build lex "king")
        kn_cat "life"))

(m49! (class (m40 (lex king))) (kn_cat life) (member b3))

(m49!)

 CPU time : 0.00

*
; The Round Table is a table
(describe
(assert member (build lex "Round Table") class (build lex "table")
        kn_cat "life"))

(m52! (class (m51 (lex table))) (kn_cat life)
 (member (m50 (lex Round Table))))
```

```
(m52!)

 CPU time : 0.01

*
; King Arthur owns the Round Table
(describe
(assert possessor *KA object (build lex "Round Table") rel (build lex "table")
        kn_cat "life"))

(m53! (kn_cat life) (object (m50 (lex Round Table))) (possessor b3)
 (rel (m51 (lex table))))

(m53!)

 CPU time : 0.01

*
; There is something named 'Excalibur'
(describe
(assert object #Excalibur proper-name (build lex "Excalibur")
        kn_cat "life"))

(m55! (kn_cat life) (object b4) (proper-name (m54 (lex Excalibur))))

(m55!)

 CPU time : 0.00

*
; Excalibur is a sword
(describe
(assert  member *Excalibur class (build lex "sword")
        kn_cat "life"))

(m57! (class (m56 (lex sword))) (kn_cat life) (member b4))

(m57!)

 CPU time : 0.01

*
; King Arthur owns Excalibur
(describe
(assert object *Excalibur rel (build lex "sword") possessor *KA
        kn_cat "life"))

(m58! (kn_cat life) (object b4) (possessor b3) (rel (m56 (lex sword))))

(m58!)

 CPU time : 0.01

*
; Something is named 'Merlin'
(describe
(assert object #Mer proper-name (build lex "Merlin")
        kn_cat "life"))
```

```
(m60! (kn_cat life) (object b5) (proper-name (m59 (lex Merlin))))

(m60!)

 CPU time : 0.02

*
; Merlin is a wizard.
(describe
(assert member *Mer  class (build lex "wizard")
        kn_cat "life"))

(m61! (class (m42 (lex wizard))) (kn_cat life) (member b5))

(m61!)

 CPU time : 0.00

*
; Something is named 'Sir Galahad'
(describe
 (assert object #Galahad proper-name (build lex "Sir Galahad")
         kn_cat "life"))

(m63! (kn_cat life) (object b6) (proper-name (m62 (lex Sir Galahad))))

(m63!)

 CPU time : 0.01

*
; Sir Galahad is a knight
(describe
(assert member *Galahad  class (build lex "knight")
        kn_cat "life"))

(m64! (class (m44 (lex knight))) (kn_cat life) (member b6))

(m64!)

 CPU time : 0.00

*
; Something is named 'Sir Tristram'
(describe
(assert object #Tris proper-name (build lex "Sir Tristram")
        kn_cat "life"))

(m66! (kn_cat life) (object b7) (proper-name (m65 (lex Sir Tristram))))

(m66!)

 CPU time : 0.01

*
; Sir Tristram is a knight
(describe
```

```
(assert member *Tris  class (build lex "knight")
        kn_cat "life"))

(m67! (class (m44 (lex knight))) (kn_cat life) (member b7))

(m67!)

 CPU time : 0.01

*
; Something is named 'Sir Gawain'
(describe
(assert object #SG proper-name (build lex "Sir Gawain")
        kn_cat "life"))

(m69! (kn_cat life) (object b8) (proper-name (m68 (lex Sir Gawain))))

(m69!)

 CPU time : 0.01

*
; Sir Gawain is a knight
(describe
(assert member *SG  class (build lex "knight")
        kn_cat "life"))

(m70! (class (m44 (lex knight))) (kn_cat life) (member b8))

(m70!)

 CPU time : 0.01

*
; Something is named 'King Ban'
(describe
(assert object #Ban proper-name (build lex "King Ban")
        kn_cat "life"))

(m72! (kn_cat life) (object b9) (proper-name (m71 (lex King Ban))))

(m72!)

 CPU time : 0.01

*
; King Ban is a king
(describe
(assert member *Ban  class (build lex "king")
        kn_cat "life"))

(m73! (class (m40 (lex king))) (kn_cat life) (member b9))

(m73!)

 CPU time : 0.01

*
```

```
; Something is named 'King Bors'
(describe
(assert object #Bors proper-name (build lex "King Bors")
        kn_cat "life"))

(m75! (kn_cat life) (object b10) (proper-name (m74 (lex King Bors))))

(m75!)

 CPU time : 0.01

*
; King Bors is a king
(describe
(assert member *Bors  class (build lex "king")
        kn_cat "life"))

(m76! (class (m40 (lex king))) (kn_cat life) (member b10))

(m76!)

 CPU time : 0.01

*
; Something is named 'King Lot'
(describe
(assert object #Lot proper-name (build lex "King Lot")
        kn_cat "life"))

(m78! (kn_cat life) (object b11) (proper-name (m77 (lex King Lot))))

(m78!)

 CPU time : 0.01

*
; King Lot is a king
(describe
(assert member *Lot  class (build lex "king")
        kn_cat "life"))

(m79! (class (m40 (lex king))) (kn_cat life) (member b11))

(m79!)

 CPU time : 0.00

*
; Sideboards are furniture
(describe
(assert subclass (build lex "sideboard") superclass (build lex "furniture")
        kn_cat "life"))

(m82! (kn_cat life) (subclass (m80 (lex sideboard)))
 (superclass (m81 (lex furniture))))

(m82!)
```

```
 CPU time : 0.01

*
; Tables are furniture
(describe
(assert subclass (build lex "table") superclass (build lex "furniture")
        kn_cat "life"))

(m83! (kn_cat life) (subclass (m51 (lex table)))
 (superclass (m81 (lex furniture))))

(m83!)

 CPU time : 0.00

*
; Chairs are furniture
(describe
(assert subclass (build lex "chair") superclass (build lex "furniture")
        kn_cat "life"))

(m85! (kn_cat life) (subclass (m84 (lex chair)))
 (superclass (m81 (lex furniture))))

(m85!)

 CPU time : 0.01

*
; Chair is a basic level category
(describe
(assert member (build lex "chair")  class (build lex "basic ctgy")
        kn_cat "life"))

(m86! (class (m18 (lex basic ctgy))) (kn_cat life) (member (m84 (lex chair))))

(m86!)

 CPU time : 0.01

*
; Table is a basic level category
(describe
(assert member (build lex "table")  class (build lex "basic ctgy")
        kn_cat "life"))

(m87! (class (m18 (lex basic ctgy))) (kn_cat life) (member (m51 (lex table))))

(m87!)

 CPU time : 0.00

*
; White is a color
(describe
(assert member (build lex "white") class (build lex "color")
        kn_cat "life"))
```

```
(m90! (class (m89 (lex color))) (kn_cat life) (member (m88 (lex white))))

(m90!)

 CPU time : 0.01

*
; Black is a color
(describe
(assert member (build lex "black") class (build lex "color")
        kn_cat "life"))

(m92! (class (m89 (lex color))) (kn_cat life) (member (m91 (lex black))))

(m92!)

 CPU time : 0.01

*
; Small is a size
(describe
(assert member (build lex "small") class (build lex "size")
        kn_cat "life"))

(m95! (class (m94 (lex size))) (kn_cat life) (member (m93 (lex small))))

(m95!)

 CPU time : 0.02

*
; "Small" and "little" are synonyms
(describe
(assert synonym (build lex "small") synonym (build lex "little")
        kn_cat "life"))

(m97! (kn_cat life) (synonym (m96 (lex little)) (m93 (lex small))))

(m97!)

 CPU time : 0.01

*
; Large is a size
(describe
(assert member (build lex "large") class (build lex "size")
        kn_cat "life"))

(m99! (class (m94 (lex size))) (kn_cat life) (member (m98 (lex large))))

(m99!)

 CPU time : 0.01

*
; "Large" and "big" are synonyms
(describe
(assert synonym (build lex "large") synonym (build lex "big")
```

```
        kn_cat "life"))

(m101! (kn_cat life) (synonym (m100 (lex big)) (m98 (lex large))))

(m101!)

 CPU time : 0.00

*
; Spears are weapons
(describe
(add subclass (build lex "spear") superclass (build lex "weapon")
        kn_cat "life"))

(m104! (kn_cat life) (subclass (m102 (lex spear)))
 (superclass (m103 (lex weapon))))

(m104!)

 CPU time : 0.01

*
; "Kill" and "Slay" are synonyms
(describe
(assert synonym (build lex "kill") synonym (build lex "slay") kn_cat "life"))

(m107! (kn_cat life) (synonym (m106 (lex slay)) (m105 (lex kill))))

(m107!)

 CPU time : 0.02

*
;#######################
;;            RULES
;#######################

; If something is a hound then that thing hunts.
(describe
(add forall $hound1
        ant (build member *hound1 class (build lex "hound"))
        cq (build agent *hound1 act (build action (build lex "hunt")))
        kn_cat "life-rule.1"))

(m110! (forall v1) (ant (p1 (class (m34 (lex hound))) (member v1)))
 (cq (p2 (act (m109 (action (m108 (lex hunt))))) (agent v1)))
 (kn_cat life-rule.1))

(m110!)

 CPU time : 0.06

*
;; If something bays and it is a member of some class then
;;  that class is a subclass of hound
(describe
 (add forall ($bayer $categ)
   &ant ((build agent *bayer act (build action (build lex "bay")))
```

```
        (build member *bayer class *categ))
   cq (build subclass *categ superclass (build lex "hound"))))

(m147! (class (m18 (lex basic ctgy))) (member (m13 (lex deer))))
(m146! (class (m18)) (member (m22 (lex horse))))
(m145! (class (m18)) (member (m29 (lex dog))))
(m144! (class (m10 (lex mammal))) (member b1))
(m143! (class (m5 (lex vertebrate))) (member b1))
(m142! (class (m4 (lex quadruped))) (member b1))
(m141! (class (m2 (lex phys obj))) (member b1))
(m140! (class (m1 (lex animal))) (member b1))
(m139! (class (m18)) (member (m38 (lex person))))
(m138! (class (m40 (lex king))) (member b3))
(m137! (class (m38)) (member b3))
(m136! (class (m81 (lex furniture))) (member (m50 (lex Round Table))))
(m135! (class (m51 (lex table))) (member (m50)))
(m134! (class (m56 (lex sword))) (member b4))
(m133! (class (m42 (lex wizard))) (member b5))
(m132! (class (m38)) (member b5))
(m131! (class (m44 (lex knight))) (member b6))
(m130! (class (m38)) (member b6))
(m129! (class (m44)) (member b7))
(m128! (class (m38)) (member b7))
(m127! (class (m44)) (member b8))
(m126! (class (m38)) (member b8))
(m125! (class (m40)) (member b9))
(m124! (class (m38)) (member b9))
(m123! (class (m40)) (member b10))
(m122! (class (m38)) (member b10))
(m121! (class (m40)) (member b11))
(m120! (class (m38)) (member b11))
(m119! (class (m18)) (member (m84 (lex chair))))
(m118! (class (m18)) (member (m51)))
(m117! (class (m89 (lex color))) (member (m88 (lex white))))
(m116! (class (m89)) (member (m91 (lex black))))
(m115! (class (m94 (lex size))) (member (m93 (lex small))))
(m114! (class (m94)) (member (m98 (lex large))))
(m113! (forall v3 v2)
 (&ant (p4 (class v3) (member v2))
  (p3 (act (m112 (action (m111 (lex bay))))) (agent v2)))
 (cq (p5 (subclass v3) (superclass (m34 (lex hound))))))
(m39! (class (m38)) (member b2))
(m36! (class (m29)) (member b1))

(m147! m146! m145! m144! m143! m142! m141! m140! m139! m138! m137! m136!
 m135! m134! m133! m132! m131! m130! m129! m128! m127! m126! m125! m124!
 m123! m122! m121! m120! m119! m118! m117! m116! m115! m114! m113! m39! m36!)

 CPU time : 0.25

*
;; Newly inferred information:
;;
;; "Deer" is a basic category
;; "Horse" is a basic category
;; "Dog" is a basic category
;; The dog (b1) is a quadruped
;; The dog (b1) is a mammal
```

```
;; The dog (b1) is a vertebrate
;; The dog (b1) is a physical object
;; The dog (b1) is a animal
;; "Person" is a basic category
;; King Arthur is a king
;; King Arthur is a person
;; The Round Table is a piece of furniture
;; The Round Table is a table
;; Excalibur is a sword
;; Merlin is a wizard
;; Merlin is a person
;; Sir Galahad is a knight
;; Sir Galahad is a person
;; Sir Tristram is a knight
;; Sir Tristram is a person
;; Sir Gawain is a knight
;; Sir Gawain is a person
;; King Ban is a king
;; King Ban is a person
;; King Bors is a king
;; King Bors is a person
;; King Lot is a king
;; King Lot is a person
;; "Chair" is a basic category
;; "Table" is a basic category
;; White is a color
;; Black is a color
;; Small is a size
;; Large is a size

;; If one thing bites another and the biter is a member of some
;;  class then that class is a subclass of animal
(describe
(add forall ($animal1 $bitten *categ)
       &ant ((build agent *animal1 act (build action (build lex "bite")
                                              object *bitten))
            (build member *animal1 class *categ))
       cq (build subclass *categ superclass (build lex "animal"))
       kn_cat "life-rule.1"))

(m149! (forall v5 v4 v3)
 (&ant (p8 (class v3) (member v4))
  (p7 (act (p6 (action (m148 (lex bite))) (object v5))) (agent v4)))
 (cq (p9 (subclass v3) (superclass (m1 (lex animal))))) (kn_cat life-rule.1))
(m147! (class (m18 (lex basic ctgy))) (member (m13 (lex deer))))
(m146! (class (m18)) (member (m22 (lex horse))))
(m145! (class (m18)) (member (m29 (lex dog))))
(m144! (class (m10 (lex mammal))) (member b1))
(m143! (class (m5 (lex vertebrate))) (member b1))
(m142! (class (m4 (lex quadruped))) (member b1))
(m141! (class (m2 (lex phys obj))) (member b1))
(m140! (class (m1)) (member b1))
(m139! (class (m18)) (member (m38 (lex person))))
(m138! (class (m40 (lex king))) (member b3))
(m137! (class (m38)) (member b3))
(m136! (class (m81 (lex furniture))) (member (m50 (lex Round Table))))
(m135! (class (m51 (lex table))) (member (m50)))
(m134! (class (m56 (lex sword))) (member b4))
```

```
(m133! (class (m42 (lex wizard))) (member b5))
(m132! (class (m38)) (member b5))
(m131! (class (m44 (lex knight))) (member b6))
(m130! (class (m38)) (member b6))
(m129! (class (m44)) (member b7))
(m128! (class (m38)) (member b7))
(m127! (class (m44)) (member b8))
(m126! (class (m38)) (member b8))
(m125! (class (m40)) (member b9))
(m124! (class (m38)) (member b9))
(m123! (class (m40)) (member b10))
(m122! (class (m38)) (member b10))
(m121! (class (m40)) (member b11))
(m120! (class (m38)) (member b11))
(m119! (class (m18)) (member (m84 (lex chair))))
(m118! (class (m18)) (member (m51)))
(m117! (class (m89 (lex color))) (member (m88 (lex white))))
(m116! (class (m89)) (member (m91 (lex black))))
(m115! (class (m94 (lex size))) (member (m93 (lex small))))
(m114! (class (m94)) (member (m98 (lex large))))
(m39! (class (m38)) (member b2))
(m36! (class (m29)) (member b1))

(m149! m147! m146! m145! m144! m143! m142! m141! m140! m139! m138! m137!
 m136! m135! m134! m133! m132! m131! m130! m129! m128! m127! m126! m125!
 m124! m123! m122! m121! m120! m119! m118! m117! m116! m115! m114! m39! m36!)

 CPU time : 0.40

*
;; Newly inferred information:
;;
;; None

;; If something is an animal and part of another class then
;;   presumably, that class is a subclass of animal
(describe
(add forall (*animal1 $class2)
       &ant ((build member *animal1 class (build lex "animal"))
             (build member *animal1 class *class2))
       cq (build mode   (build lex "presumably")
                 object (build subclass *class2 superclass (build lex "animal")))
       kn_cat "life-rule.1"))

(m173! (mode (m150 (lex presumably)))
 (object
  (m172 (subclass (m5 (lex vertebrate))) (superclass (m1 (lex animal))))))
(m171! (mode (m150))
 (object (m170 (subclass (m4 (lex quadruped))) (superclass (m1)))))
(m167! (mode (m150))
 (object (m166 (subclass (m10 (lex mammal))) (superclass (m1)))))
(m165! (mode (m150))
 (object (m164 (subclass (m13 (lex deer))) (superclass (m1)))))
(m163! (mode (m150))
 (object (m162 (subclass (m20 (lex hart))) (superclass (m1)))))
(m161! (mode (m150))
 (object (m160 (subclass (m22 (lex horse))) (superclass (m1)))))
(m157! (mode (m150))
```

```
 (object (m156 (subclass (m29 (lex dog))) (superclass (m1)))))
(m155! (mode (m150))
 (object (m154 (subclass (m34 (lex hound))) (superclass (m1)))))
(m153! (mode (m150))
 (object (m152 (subclass (m2 (lex phys obj))) (superclass (m1)))))
(m151! (forall v6 v4)
 (&ant (p11 (class v6) (member v4)) (p10 (class (m1)) (member v4)))
 (cq (p13 (mode (m150)) (object (p12 (subclass v6) (superclass (m1))))))
 (kn_cat life-rule.1))
(m147! (class (m18 (lex basic ctgy))) (member (m13)))
(m146! (class (m18)) (member (m22)))
(m145! (class (m18)) (member (m29)))
(m144! (class (m10)) (member b1))
(m143! (class (m5)) (member b1))
(m142! (class (m4)) (member b1))
(m141! (class (m2)) (member b1))
(m140! (class (m1)) (member b1))
(m139! (class (m18)) (member (m38 (lex person))))
(m138! (class (m40 (lex king))) (member b3))
(m137! (class (m38)) (member b3))
(m136! (class (m81 (lex furniture))) (member (m50 (lex Round Table))))
(m135! (class (m51 (lex table))) (member (m50)))
(m134! (class (m56 (lex sword))) (member b4))
(m133! (class (m42 (lex wizard))) (member b5))
(m132! (class (m38)) (member b5))
(m131! (class (m44 (lex knight))) (member b6))
(m130! (class (m38)) (member b6))
(m129! (class (m44)) (member b7))
(m128! (class (m38)) (member b7))
(m127! (class (m44)) (member b8))
(m126! (class (m38)) (member b8))
(m125! (class (m40)) (member b9))
(m124! (class (m38)) (member b9))
(m123! (class (m40)) (member b10))
(m122! (class (m38)) (member b10))
(m121! (class (m40)) (member b11))
(m120! (class (m38)) (member b11))
(m119! (class (m18)) (member (m84 (lex chair))))
(m118! (class (m18)) (member (m51)))
(m117! (class (m89 (lex color))) (member (m88 (lex white))))
(m116! (class (m89)) (member (m91 (lex black))))
(m115! (class (m94 (lex size))) (member (m93 (lex small))))
(m114! (class (m94)) (member (m98 (lex large))))
(m39! (class (m38)) (member b2))
(m36! (class (m29)) (member b1))

(m173! m171! m167! m165! m163! m161! m157! m155! m153! m151! m147! m146!
 m145! m144! m143! m142! m141! m140! m139! m138! m137! m136! m135! m134!
 m133! m132! m131! m130! m129! m128! m127! m126! m125! m124! m123! m122!
 m121! m120! m119! m118! m117! m116! m115! m114! m39! m36!)

 CPU time : 0.80

*
;; Newly inferred information:
;;
;; Presumably, quadruped is a subclass of animal
;; Presumably, vertebrate is a subclass of animal
```

```
;; Presumably, mammal is a subclass of animal
;; Presumably, deer is a subclass of animal
;; Presumably, hart is a subclass of animal
;; Presumably, horse is a subclass of animal
;; Presumably, dog is a subclass of animal
;; Presumably, hound is a subclass of animal
;; Presumably, physical object is a subclass of animal

;; If something is presumably an animal and is a member of another class then
;;  presumably, that class is a subclass of animal
(describe
(add forall (*animal1 *class2)
        &ant ((build mode (build lex "presumably")
                     object (build member *animal1 class (build lex "animal")))
              (build member *animal1  class *class2))
        cq (build mode (build lex "presumably")
                  object (build subclass *class2
                                 superclass (build lex "animal")))
        kn_cat "life-rule.1"))

(m174! (forall v6 v4)
 (&ant
  (p15 (mode (m150 (lex presumably)))
   (object (p10 (class (m1 (lex animal))) (member v4))))
  (p11 (class v6) (member v4)))
 (cq (p13 (mode (m150)) (object (p12 (subclass v6) (superclass (m1))))))
 (kn_cat life-rule.1))

(m174!)

 CPU time : 0.07

*
;; Newly inferred information:
;;
;; None

;; If something is a mammal and a member of another class then
;; presumably, that class is a subclass of mammal
(describe
(add forall (*animal1 *class2)
        &ant ((build member *animal1  class (build lex "mammal"))
              (build member *animal1  class *class2))
        cq (build mode   (build lex "presumably")
                  object (build subclass *class2 superclass (build lex "mammal")))
        kn_cat "life-rule.1"))

(m197! (mode (m150 (lex presumably)))
 (object (m196 (subclass (m10 (lex mammal))) (superclass (m10)))))
(m195! (mode (m150))
 (object (m194 (subclass (m13 (lex deer))) (superclass (m10)))))
(m193! (mode (m150))
 (object (m192 (subclass (m20 (lex hart))) (superclass (m10)))))
(m191! (mode (m150))
 (object (m190 (subclass (m22 (lex horse))) (superclass (m10)))))
(m189! (mode (m150))
 (object (m188 (subclass (m27 (lex pony))) (superclass (m10)))))
(m187! (mode (m150))
```

```
 (object (m186 (subclass (m34 (lex hound))) (superclass (m10)))))
(m185! (mode (m150))
 (object (m184 (subclass (m1 (lex animal))) (superclass (m10)))))
(m183! (mode (m150))
 (object (m182 (subclass (m4 (lex quadruped))) (superclass (m10)))))
(m181! (mode (m150))
 (object (m180 (subclass (m29 (lex dog))) (superclass (m10)))))
(m179! (mode (m150))
 (object (m178 (subclass (m5 (lex vertebrate))) (superclass (m10)))))
(m177! (mode (m150))
 (object (m176 (subclass (m2 (lex phys obj))) (superclass (m10)))))
(m175! (forall v6 v4)
 (&ant (p16 (class (m10)) (member v4)) (p11 (class v6) (member v4)))
 (cq (p18 (mode (m150)) (object (p17 (subclass v6) (superclass (m10))))))
 (kn_cat life-rule.1))
(m144! (class (m10)) (member b1))

(m197! m195! m193! m191! m189! m187! m185! m183! m181! m179! m177! m175!
 m144!)

 CPU time : 0.58

*
;; Newly inferred information:
;;
;; Presumably, mammal is a subclass of mammal
;; Presumably, deer is a subclass of mammal
;; Presumably, hart is a subclass of mammal
;; Presumably, horse is a subclass of mammal
;; Presumably, pony is a subclass of mammal
;; Presumably, hound is a subclass of mammal
;; Presumably, animal is a subclass of mammal
;; Presumably, vertebrate is a subclass of mammal
;; Presumably, quadruped is a subclass of mammal
;; Presumably, dog is a subclass of mammal
;; Presumably, physical object is a subclass of mammal

;; If something is a mammal, then it presumably bears live young
(describe
(add forall *animal1
        ant (build member *animal1  class (build lex "mammal"))
        cq (build mode (build lex "presumably")
                  object (build agent *animal1 act (build action (build lex "bear")
                                                          object (build lex "live young"))))
        kn_cat "life-rule.1"))

(m203! (mode (m150 (lex presumably)))
 (object
  (m202
   (act (m200 (action (m198 (lex bear))) (object (m199 (lex live young)))))
   (agent b1))))
(m201! (forall v4) (ant (p16 (class (m10 (lex mammal))) (member v4)))
 (cq (p21 (mode (m150)) (object (p20 (act (m200)) (agent v4)))))
 (kn_cat life-rule.1))

(m203! m201!)

 CPU time : 0.06
```

```
*
;; Newly inferred information:
;;
;; The dog (b1) bears live young

;"If something bears something else, the bearer is an animal"
(describe
(add forall (*animal1 $animal2)
        ant (build agent *animal1 act (build action (build lex "bear")
                                            object *animal2))
        cq (build member *animal1  class (build lex "mammal"))
        kn_cat "life-rule.1"))

(m204! (forall v7 v4)
 (ant (p23 (act (p22 (action (m198 (lex bear))) (object v7))) (agent v4)))
 (cq (p16 (class (m10 (lex mammal))) (member v4))) (kn_cat life-rule.1))

(m204!)

 CPU time : 0.05

*
;; Newly inferred information:
;;
;; None

; If there is a person and that person can carry something, then the
;  thing that can be carried has the property "small".
(describe
(add forall ($thingy $person)
        &ant ((build member *person class (build lex "person"))
              (build agent *person act (build action (build lex "carry") object *thingy)))
        cq (build object *thingy property (build lex "small"))
        kn_cat "life-rule.2"))

(m206! (forall v9 v8)
 (&ant (p26 (act (p25 (action (m205 (lex carry))) (object v8))) (agent v9))
  (p24 (class (m38 (lex person))) (member v9)))
 (cq (p27 (object v8) (property (m93 (lex small))))) (kn_cat life-rule.2))
(m137! (class (m38)) (member b3))
(m132! (class (m38)) (member b5))
(m130! (class (m38)) (member b6))
(m128! (class (m38)) (member b7))
(m126! (class (m38)) (member b8))
(m124! (class (m38)) (member b9))
(m122! (class (m38)) (member b10))
(m120! (class (m38)) (member b11))
(m39! (class (m38)) (member b2))

(m206! m137! m132! m130! m128! m126! m124! m122! m120! m39!)

 CPU time : 0.37

*
;; Newly inferred information:
;;
;; None
```

```
; If something wants something then the thing that is wanted is valuable
(describe
(add forall (*thingy *person)
        ant (build agent *person act (build action (build lex "want") object *thingy))
        cq (build object *thingy property (build lex "valuable"))
        kn_cat "life-rule.2"))

(m209! (forall v9 v8)
 (ant (p31 (act (p30 (action (m207 (lex want))) (object v8))) (agent v9)))
 (cq (p32 (object v8) (property (m208 (lex valuable))))) (kn_cat life-rule.2))

(m209!)

 CPU time : 0.08

*
;; Newly inferred information:
;;
;; None

; If something says that it wants another thing, then it actually
;  does want that thing
(describe
(add forall (*thingy *person)
        ant (build agent *person act (build action (build lex "say that")
                                           object (build agent *person
                                                         act (build action (build lex "want")
                                                                   object *thingy))))
        cq (build agent *person
                  act (build action (build lex "want")
                             object *thingy))
        kn_cat "life-rule.2"))

(m211! (forall v9 v8)
 (ant
  (p34
   (act (p33 (action (m210 (lex say that)))
         (object
          (p31 (act (p30 (action (m207 (lex want))) (object v8)))
           (agent v9)))))
   (agent v9)))
 (cq (p31)) (kn_cat life-rule.2))

(m211!)

 CPU time : 0.05

*
;; Newly inferred information:
;;
;; None.

;
(describe
; If a member of some class has a property that is a color,
;  then the class that it is a member of is a subclass of 'physical object'
(add forall ($thing $prop $foo)
```

```
        &ant ((build member *foo class *thing)
              (build object *foo property *prop)
              (build member *prop class (build lex "color")))
        cq (build subclass *thing superclass (build lex "phys obj"))
        kn_cat "intrinsic"))

(m212! (forall v12 v11 v10)
 (&ant (p37 (class (m89 (lex color))) (member v11))
  (p36 (object v12) (property v11)) (p35 (class v10) (member v12)))
 (cq (p38 (subclass v10) (superclass (m2 (lex phys obj)))))
 (kn_cat intrinsic))
(m147! (class (m18 (lex basic ctgy))) (member (m13 (lex deer))))
(m146! (class (m18)) (member (m22 (lex horse))))
(m145! (class (m18)) (member (m29 (lex dog))))
(m144! (class (m10 (lex mammal))) (member b1))
(m143! (class (m5 (lex vertebrate))) (member b1))
(m142! (class (m4 (lex quadruped))) (member b1))
(m141! (class (m2)) (member b1))
(m140! (class (m1 (lex animal))) (member b1))
(m139! (class (m18)) (member (m38 (lex person))))
(m138! (class (m40 (lex king))) (member b3))
(m137! (class (m38)) (member b3))
(m136! (class (m81 (lex furniture))) (member (m50 (lex Round Table))))
(m135! (class (m51 (lex table))) (member (m50)))
(m134! (class (m56 (lex sword))) (member b4))
(m133! (class (m42 (lex wizard))) (member b5))
(m132! (class (m38)) (member b5))
(m131! (class (m44 (lex knight))) (member b6))
(m130! (class (m38)) (member b6))
(m129! (class (m44)) (member b7))
(m128! (class (m38)) (member b7))
(m127! (class (m44)) (member b8))
(m126! (class (m38)) (member b8))
(m125! (class (m40)) (member b9))
(m124! (class (m38)) (member b9))
(m123! (class (m40)) (member b10))
(m122! (class (m38)) (member b10))
(m121! (class (m40)) (member b11))
(m120! (class (m38)) (member b11))
(m119! (class (m18)) (member (m84 (lex chair))))
(m118! (class (m18)) (member (m51)))
(m117! (class (m89)) (member (m88 (lex white))))
(m116! (class (m89)) (member (m91 (lex black))))
(m115! (class (m94 (lex size))) (member (m93 (lex small))))
(m114! (class (m94)) (member (m98 (lex large))))
(m39! (class (m38)) (member b2))
(m36! (class (m29)) (member b1))

(m212! m147! m146! m145! m144! m143! m142! m141! m140! m139! m138! m137!
 m136! m135! m134! m133! m132! m131! m130! m129! m128! m127! m126! m125!
 m124! m123! m122! m121! m120! m119! m118! m117! m116! m115! m114! m39! m36!)

 CPU time : 0.77

*
;; Newly inferred information:
;;
;; None.
```

```
; If a member of some class has a property that is a size,
;  then the class that it is a member of is a subclass of 'physical object'
(describe
(add forall (*thing *prop *foo)
        &ant ((build member *foo class *thing)
              (build object *foo property *prop)
              (build member *prop class (build lex "size")))
        cq (build subclass *thing superclass (build lex "phys obj"))
        kn_cat "intrinsic"))

(m213! (forall v12 v11 v10)
 (&ant (p42 (class (m94 (lex size))) (member v11))
  (p36 (object v12) (property v11)) (p35 (class v10) (member v12)))
 (cq (p38 (subclass v10) (superclass (m2 (lex phys obj)))))
 (kn_cat intrinsic))
(m115! (class (m94)) (member (m93 (lex small))))
(m114! (class (m94)) (member (m98 (lex large))))

(m213! m115! m114!)

 CPU time : 0.14

*
;; Newly inferred information:
;;
;; None.

; A weapon damages
(describe
(add forall $weapon1
        ant (build member *weapon1  class (build lex "weapon"))
        cq (build agent *weapon1 act (build action (build lex "damage")))
        kn_cat "life-rule.1"))

(m216! (forall v13) (ant (p46 (class (m103 (lex weapon))) (member v13)))
 (cq (p47 (act (m215 (action (m214 (lex damage))))) (agent v13)))
 (kn_cat life-rule.1))

(m216!)

 CPU time : 0.09

*
;; Newly inferred information:
;;
;; None.

; If something is an elder then that thing is old and is presumably a person
(describe
(add forall $eld1
        ant (build member *eld1  class (build lex "elder"))
        cq ((build object *eld1 property (build lex "old"))
            (build mode (build lex "presumably")
                   object (build member *eld1 class (build lex "person"))))
        kn_cat "life-rule.1"))

(m219! (forall v14) (ant (p51 (class (m217 (lex elder))) (member v14)))
```

```
 (cq
  (p54 (mode (m150 (lex presumably)))
   (object (p53 (class (m38 (lex person))) (member v14))))
  (p52 (object v14) (property (m218 (lex old)))))
 (kn_cat life-rule.1))

(m219!)

 CPU time : 0.12

*
;; Newly inferred information:
;;
;; None.

; if one thing chases another, the former runs behind the latter
(describe
(add forall ($chaser $chasee)
        ant (build agent *chaser act (build action (build lex "chase") object *chasee))
        cq ((build agent *chaser act (build action (build lex "run")))
            (build object1 *chaser rel (build lex "behind") object2 *chasee))
        kn_cat "life-rule.1"))

(m224! (forall v16 v15)
 (ant (p59 (act (p58 (action (m220 (lex chase))) (object v16))) (agent v15)))
 (cq (p61 (object1 v15) (object2 v16) (rel (m223 (lex behind))))
  (p60 (act (m222 (action (m221 (lex run))))) (agent v15)))
 (kn_cat life-rule.1))

(m224!)

 CPU time : 0.06

*
;; Newly inferred information:
;;
;; None.

End of /projects/stn2/CVA/demos/brachet.base demonstration.

 CPU time : 4.86

*
;;; This is the content of Karen's original "br.txt1" file  ;;;

;Right so as they sat, there came a white hart running into the hall with
;a white brachet next to him, and thirty couples of black hounds came
;running after them with a great cry.

;; In the story, there is a hart
(describe
(add member #hart1  class (build lex "hart")
        kn_cat "story"))

(m239! (mode (m150 (lex presumably)))
 (object
  (m238
   (act (m200 (action (m198 (lex bear))) (object (m199 (lex live young)))))
```

```
   (agent b12))))
(m237! (mode (m150))
 (object
  (m236 (subclass (m8 (lex herbivore))) (superclass (m10 (lex mammal))))))
(m235! (mode (m150))
 (object (m234 (subclass (m8)) (superclass (m1 (lex animal))))))
(m233! (class (m20 (lex hart))) (member b12))
(m232! (class (m13 (lex deer))) (member b12))
(m231! (class (m10)) (member b12))
(m230! (class (m8)) (member b12))
(m229! (class (m5 (lex vertebrate))) (member b12))
(m228! (class (m4 (lex quadruped))) (member b12))
(m227! (class (m2 (lex phys obj))) (member b12))
(m226! (class (m1)) (member b12))
(m225! (class (m20)) (kn_cat story) (member b12))
(m197! (mode (m150)) (object (m196 (subclass (m10)) (superclass (m10)))))
(m195! (mode (m150)) (object (m194 (subclass (m13)) (superclass (m10)))))
(m193! (mode (m150)) (object (m192 (subclass (m20)) (superclass (m10)))))
(m191! (mode (m150))
 (object (m190 (subclass (m22 (lex horse))) (superclass (m10)))))
(m189! (mode (m150))
 (object (m188 (subclass (m27 (lex pony))) (superclass (m10)))))
(m187! (mode (m150))
 (object (m186 (subclass (m34 (lex hound))) (superclass (m10)))))
(m185! (mode (m150)) (object (m184 (subclass (m1)) (superclass (m10)))))
(m183! (mode (m150)) (object (m182 (subclass (m4)) (superclass (m10)))))
(m181! (mode (m150))
 (object (m180 (subclass (m29 (lex dog))) (superclass (m10)))))
(m179! (mode (m150)) (object (m178 (subclass (m5)) (superclass (m10)))))
(m177! (mode (m150)) (object (m176 (subclass (m2)) (superclass (m10)))))
(m173! (mode (m150)) (object (m172 (subclass (m5)) (superclass (m1)))))
(m171! (mode (m150)) (object (m170 (subclass (m4)) (superclass (m1)))))
(m167! (mode (m150)) (object (m166 (subclass (m10)) (superclass (m1)))))
(m165! (mode (m150)) (object (m164 (subclass (m13)) (superclass (m1)))))
(m163! (mode (m150)) (object (m162 (subclass (m20)) (superclass (m1)))))
(m161! (mode (m150)) (object (m160 (subclass (m22)) (superclass (m1)))))
(m157! (mode (m150)) (object (m156 (subclass (m29)) (superclass (m1)))))
(m155! (mode (m150)) (object (m154 (subclass (m34)) (superclass (m1)))))
(m153! (mode (m150)) (object (m152 (subclass (m2)) (superclass (m1)))))

(m239! m237! m235! m233! m232! m231! m230! m229! m228! m227! m226! m225!
 m197! m195! m193! m191! m189! m187! m185! m183! m181! m179! m177! m173!
 m171! m167! m165! m163! m161! m157! m155! m153!)

 CPU time : 0.95

*
;; Newly inferred information:
;;
;; Presumably, the hart bears live young
;; Presumably, the herbivores are mammals
;; Presumably, herbivores are animals
;; The hart is a hart
;; The hart is a quadruped
;; The hart is a deer
;; The hart is a mammal
;; The hart is a herbivore
;; The hart is a vertebrate
```

```
;; The hart is a physical object
;; The hart is a animal

;; In the story, the hart runs into something
(describe
(add agent *hart1 act (build action (build lex "run")) into #hall1 kn_cat "story"))

(m245! (act (m222 (action (m221 (lex run))))) (agent b12))
(m244! (act (m222)) (agent b12) (into b13) (kn_cat story))

(m245! m244!)

 CPU time : 0.04

*
;; Newly inferred information:
;;
;; None.

;; In the story, the thing that the hart runs into is a hall
(describe
(add member *hall1 class (build lex "hall") kn_cat "story"))

(m248! (class (m246 (lex hall))) (member b13))
(m247! (class (m246)) (kn_cat story) (member b13))

(m248! m247!)

 CPU time : 0.05

*
;; Newly inferred information:
;;
;; None.

;; In the story, the hall is King Arthur's hall (KA from background)
(describe
(add object *hall1 rel (build lex "hall") possessor *KA kn_cat "story-comp"))

(m249! (kn_cat story-comp) (object b13) (possessor b3)
 (rel (m246 (lex hall))))

(m249!)

 CPU time : 0.05

*
;; Newly inferred information:
;;
;; None.

;; In the story, there is a brachet
(describe
(add member #brachet1  class (build lex "brachet")
        kn_cat "story"))

(m252! (class (m250 (lex brachet))) (member b14))
(m251! (class (m250)) (kn_cat story) (member b14))
```

```
(m252! m251!)

 CPU time : 0.05

*
;; Newly inferred information:
;;
;; None.

;; In the story, the brachet is next to a hart
(describe
 (add object *brachet1 location
      (build sp-rel (build lex "next to") object (build lex "hart"))
      kn_cat "story"))

(m255! (kn_cat story)
 (location (m254 (object (m20 (lex hart))) (sp-rel (m253 (lex next to)))))
 (object b14))

(m255!)

 CPU time : 0.04

*
;; Newly inferred information:
;;
;; None.

;; In the story, the brachet is white
(describe
(add object *brachet1 property (build lex "white") kn_cat "story"))

(m258! (subclass (m250 (lex brachet))) (superclass (m2 (lex phys obj))))
(m257! (object b14) (property (m88 (lex white))))
(m256! (kn_cat story) (object b14) (property (m88)))

(m258! m257! m256!)

 CPU time : 0.07

*
;; Newly inferred information:
;;
;; Brachet is a subclass of physical object.

;; What does brachet mean?
^
-->(defineNoun "brachet")
 Definition of brachet:
 Class Inclusions: phys obj,
 Possible Properties: white,
 Possibly Similar Items: animal, mammal, deer, horse, pony, dog,
nil
 CPU time : 8.21

*
;; In the story, there is a hound
```

```
(describe
(add member #hounds1  class (build lex "hound") kn_cat "story"))

(m305! (mode (m150 (lex presumably)))
 (object
  (m304
   (act (m200 (action (m198 (lex bear))) (object (m199 (lex live young)))))
   (agent b15))))
(m303! (act (m109 (action (m108 (lex hunt))))) (agent b15))
(m302! (class (m34 (lex hound))) (member b15))
(m301! (class (m29 (lex dog))) (member b15))
(m300! (class (m10 (lex mammal))) (member b15))
(m299! (class (m5 (lex vertebrate))) (member b15))
(m298! (class (m4 (lex quadruped))) (member b15))
(m297! (class (m2 (lex phys obj))) (member b15))
(m296! (class (m1 (lex animal))) (member b15))
(m295! (class (m34)) (kn_cat story) (member b15))
(m264! (mode (m150))
 (object (m261 (subclass (m250 (lex brachet))) (superclass (m10)))))
(m263! (mode (m150)) (object (m260 (subclass (m250)) (superclass (m1)))))
(m197! (mode (m150)) (object (m196 (subclass (m10)) (superclass (m10)))))
(m195! (mode (m150))
 (object (m194! (subclass (m13 (lex deer))) (superclass (m10)))))
(m193! (mode (m150))
 (object (m192 (subclass (m20 (lex hart))) (superclass (m10)))))
(m191! (mode (m150))
 (object (m190 (subclass (m22 (lex horse))) (superclass (m10)))))
(m189! (mode (m150))
 (object (m188 (subclass (m27 (lex pony))) (superclass (m10)))))
(m187! (mode (m150)) (object (m186 (subclass (m34)) (superclass (m10)))))
(m185! (mode (m150)) (object (m184 (subclass (m1)) (superclass (m10)))))
(m183! (mode (m150)) (object (m182 (subclass (m4)) (superclass (m10)))))
(m181! (mode (m150)) (object (m180! (subclass (m29)) (superclass (m10)))))
(m179! (mode (m150)) (object (m178 (subclass (m5)) (superclass (m10)))))
(m177! (mode (m150)) (object (m176 (subclass (m2)) (superclass (m10)))))
(m173! (mode (m150)) (object (m172 (subclass (m5)) (superclass (m1)))))
(m171! (mode (m150)) (object (m170 (subclass (m4)) (superclass (m1)))))
(m169! (mode (m150)) (object (m168 (subclass (m1)) (superclass (m1)))))
(m167! (mode (m150)) (object (m166! (subclass (m10)) (superclass (m1)))))
(m165! (mode (m150)) (object (m164! (subclass (m13)) (superclass (m1)))))
(m163! (mode (m150)) (object (m162 (subclass (m20)) (superclass (m1)))))
(m161! (mode (m150)) (object (m160! (subclass (m22)) (superclass (m1)))))
(m159! (mode (m150)) (object (m158! (subclass (m27)) (superclass (m1)))))
(m157! (mode (m150)) (object (m156! (subclass (m29)) (superclass (m1)))))
(m155! (mode (m150)) (object (m154 (subclass (m34)) (superclass (m1)))))
(m153! (mode (m150)) (object (m152 (subclass (m2)) (superclass (m1)))))

(m305! m303! m302! m301! m300! m299! m298! m297! m296! m295! m264! m263!
 m197! m195! m193! m191! m189! m187! m185! m183! m181! m179! m177! m173!
 m171! m169! m167! m165! m163! m161! m159! m157! m155! m153!)

 CPU time : 1.84

*
;; Newly inferred information:
;;
;; Presumably, the hound bears live young
;; The hound hunts.
```

```
;; The hound is a hound.
;; The hound is a dog.
;; The hound is a quadruped.
;; The hound is a mammal.
;; The hound is a vertebrate.
;; The hound is a physical object.
;; The hound is an animal.
;; In the story, the hound is a hound.
;; Presumably, brachet is a subclass of mammal.
;; Presumably, brachet is a subclass of animal

;; In the story, the hound is black
(describe
(add object *hounds1 property (build lex "black") kn_cat "story"))

(m312! (subclass (m20 (lex hart))) (superclass (m2 (lex phys obj))))
(m311! (subclass (m34 (lex hound))) (superclass (m2)))
(m310! (subclass (m5 (lex vertebrate))) (superclass (m2)))
(m309! (subclass (m4 (lex quadruped))) (superclass (m2)))
(m308! (subclass (m2)) (superclass (m2)))
(m307! (object b15) (property (m91 (lex black))))
(m306! (kn_cat story) (object b15) (property (m91)))
(m285! (subclass (m29 (lex dog))) (superclass (m2)))
(m283! (subclass (m27 (lex pony))) (superclass (m2)))
(m278! (subclass (m22 (lex horse))) (superclass (m2)))
(m273! (subclass (m13 (lex deer))) (superclass (m2)))
(m270! (subclass (m10 (lex mammal))) (superclass (m2)))
(m268! (subclass (m1 (lex animal))) (superclass (m2)))
(m258! (subclass (m250 (lex brachet))) (superclass (m2)))

(m312! m311! m310! m309! m308! m307! m306! m285! m283! m278! m273! m270!
 m268! m258!)

 CPU time : 2.09

*
;; Newly inferred information:
;;
;; Hart is a subclass of physical object
;; Hound is a subclass of physical object
;; Vertebrate is a subclass of physical object
;; Quadruped is a subclass of physical object
;; Physical object is a subclass of physical object
;; Dog is a subclass of physical object
;; Pony is a subclass of physical object
;; Horse is a subclass of physical object
;; Deer is a subclass of physical object
;; Mammal is a subclass of physical object
;; Animal is a subclass of physical object

;; In the story, the hound is running
(describe
(add agent *hounds1 act (build action (build lex "run")) kn_cat "story"))

(m314! (act (m222 (action (m221 (lex run))))) (agent b15))
(m313! (act (m222)) (agent b15) (kn_cat story))

(m314! m313!)
```

```
 CPU time : 0.05

*
;; Newly inferred information:
;;
;; None.

;; In the story, the hound is behind the hart
(describe
(add object *hounds1
     location (build sp-rel (build lex "behind")
                     object *hart1)
     kn_cat "story"))

(m316! (kn_cat story)
 (location (m315 (object b12) (sp-rel (m223 (lex behind))))) (object b15))

(m316!)

 CPU time : 0.05

*
;; Newly inferred information:
;;
;; None.

;Then the hart went running about the Round Table; as he went by the sideboard,
;the white brachet bit him in the buttock [. . .]

;; In the story, the hart is next to the Round Table.
(describe
(add object *hart1
        location (build sp-rel (build lex "next to")
                        object (build lex "Round Table"))
        kn_cat "story"))

(m318! (kn_cat story)
 (location
  (m317 (object (m50 (lex Round Table))) (sp-rel (m253 (lex next to)))))
 (object b12))

(m318!)

 CPU time : 0.04

*
;; Newly inferred information:
;;
;; None.

;; In the story, the brachet bites something.
(describe
(add agent *brachet1
     act (build action (build lex "bite")
                object #butt1)
     kn_cat "story"))
```

```
(m321! (act (m319 (action (m148 (lex bite))) (object b16))) (agent b14))
(m320! (act (m319)) (agent b14) (kn_cat story))
(m260! (subclass (m250 (lex brachet))) (superclass (m1 (lex animal))))

(m321! m320! m260!)

 CPU time : 0.05

*
;; Newly inferred information:
;;
;; Brachet is a subclass of animal

;; In the story, the thing that the brachet bites is part of the hart.
(describe
(add whole *hart1 part *butt1 kn_cat "story"))

(m322! (kn_cat story) (part b16) (whole b12))

(m322!)

 CPU time : 0.05

*
;; Newly inferred information:
;;
;; None.

;; In the story, the thing that the brachet bites is a buttock
(describe
(add member *butt1 class (build lex "buttock") kn_cat "story"))

(m325! (class (m323 (lex buttock))) (member b16))
(m324! (class (m323)) (kn_cat story) (member b16))

(m325! m324!)

 CPU time : 0.08

*
;; Newly inferred information:
;;
;; None.

;; What does brachet mean?
^
-->(defineNoun "brachet")
 Definition of brachet:
 Class Inclusions: animal,
 Possible Actions: bite buttock,
 Possible Properties: white,
 Possibly Similar Items: mammal, pony,
nil
 CPU time : 5.69

* ;
;Therewith the knight arose, took up the brachet, went forth out of the hall,
;took his horse and rode away with the brachet.
```

```
;; In the story, there is a knight
(describe
(add member #knight1 class (build lex "knight") kn_cat "story"))

(m328! (class (m44 (lex knight))) (member b17))
(m327! (class (m38 (lex person))) (member b17))
(m326! (class (m44)) (kn_cat story) (member b17))

(m328! m327! m326!)

 CPU time : 0.12

*
;; Newly inferred information:
;;
;; The knight is a knight
;; The knight is a person

;; In the story, the knight arises
(describe
(add agent *knight1 act (build action (build lex "arise")) kn_cat "story"))

(m331! (act (m330 (action (m329 (lex arise))))) (agent b17) (kn_cat story))

(m331!)

 CPU time : 0.06

*
;; Newly inferred information:
;;
;; None.

;; In the story, the knight picks up the brachet
(describe
(add agent *knight1
     act (build action (build lex "pick up")
                object *brachet1)
     kn_cat "story"))

(m334! (act (m333 (action (m332 (lex pick up))) (object b14))) (agent b17)
 (kn_cat story))

(m334!)

 CPU time : 0.06

*
;; Newly inferred information:
;;
;; None.

;; In the story, the knight mounts something
(describe
(add agent *knight1
     act (build action (build lex "mount")
                object #horsey1)
```

```
     kn_cat "story"))

(m337! (act (m336 (action (m335 (lex mount))) (object b18))) (agent b17)
 (kn_cat story))

(m337!)

 CPU time : 0.07

*
;; Newly inferred information:
;;
;; None.

;; In the story, the thing that the knight mounts is a horse
(describe
(add member *horsey1 class (build lex "horse") kn_cat "story"))

(m344! (class (m22 (lex horse))) (member b18))
(m343! (class (m8 (lex herbivore))) (member b18))
(m342! (class (m5 (lex vertebrate))) (member b18))
(m341! (class (m4 (lex quadruped))) (member b18))
(m340! (class (m2 (lex phys obj))) (member b18))
(m339! (class (m1 (lex animal))) (member b18))
(m338! (class (m22)) (kn_cat story) (member b18))
(m263! (mode (m150 (lex presumably)))
 (object (m260! (subclass (m250 (lex brachet))) (superclass (m1)))))
(m235! (mode (m150)) (object (m234 (subclass (m8)) (superclass (m1)))))
(m173! (mode (m150)) (object (m172 (subclass (m5)) (superclass (m1)))))
(m171! (mode (m150)) (object (m170 (subclass (m4)) (superclass (m1)))))
(m169! (mode (m150)) (object (m168 (subclass (m1)) (superclass (m1)))))
(m167! (mode (m150))
 (object (m166! (subclass (m10 (lex mammal))) (superclass (m1)))))
(m165! (mode (m150))
 (object (m164! (subclass (m13 (lex deer))) (superclass (m1)))))
(m163! (mode (m150))
 (object (m162 (subclass (m20 (lex hart))) (superclass (m1)))))
(m161! (mode (m150)) (object (m160! (subclass (m22)) (superclass (m1)))))
(m159! (mode (m150))
 (object (m158! (subclass (m27 (lex pony))) (superclass (m1)))))
(m157! (mode (m150))
 (object (m156! (subclass (m29 (lex dog))) (superclass (m1)))))
(m155! (mode (m150))
 (object (m154 (subclass (m34 (lex hound))) (superclass (m1)))))
(m153! (mode (m150)) (object (m152 (subclass (m2)) (superclass (m1)))))

(m344! m343! m342! m341! m340! m339! m338! m263! m235! m173! m171! m169!
 m167! m165! m163! m161! m159! m157! m155! m153!)

 CPU time : 1.61

*
;; Newly inferred information:
;;
;; The horse is a horse.
;; The horse is a herbivore.
;; The horse is a vertebrate.
;; The horse is a quadruped.
```

```
;; The horse is a physical object.
;; The horse is an animal.
;;

;; In the story, the knight rides the horse
(describe
(add agent *knight1
     act (build action (build lex "ride")
               object *horsey1)
      kn_cat "story"))

(m347! (act (m346 (action (m345 (lex ride))) (object b18))) (agent b17)
 (kn_cat story))

(m347!)

 CPU time : 0.07

*
;; Newly inferred information:
;;
;; None.

;; In the story, the knight carries the brachet
(describe
(add agent *knight1
     act (build action (build lex "carry")
               object *brachet1)
     kn_cat "story"))

(m351! (object b14) (property (m93 (lex small))))
(m350! (act (m348 (action (m205 (lex carry))) (object b14))) (agent b17))
(m349! (act (m348)) (agent b17) (kn_cat story))
(m258! (subclass (m250 (lex brachet))) (superclass (m2 (lex phys obj))))

(m351! m350! m349! m258!)

 CPU time : 0.16

*
;; Newly inferred information:
;;
;; The brachet is small.

;; What does brachet mean?
^
-->(defineNoun "brachet")
 Definition of brachet:
 Class Inclusions: animal,
 Possible Actions: bite buttock,
 Possible Properties: small, white,
 Possibly Similar Items: mammal, pony,
nil
 CPU time : 7.14

*
;
;Right so a lady came in on a white palfrey and cried aloud to King Arthur,
```

```
;‘Sire, suffer me not to have this spite for the brachet is mine that the knight
;led away.’”  [p66]

;; In the story, there is a lady
(describe
(add member #lady1  class (build lex ”lady”) kn_cat ”story”))

(m354! (class (m352 (lex lady))) (member b19))
(m353! (class (m352)) (kn_cat story) (member b19))

(m354! m353!)

 CPU time : 0.08

*
;; Newly inferred information:
;;
;; None.

;; In the story, the lady says that the knight took the brachet
(describe
(add agent *lady1
     act (build action (build lex ”say that”)
                object (build agent *knight1
                              act (build action (build lex ”take”)
                                        object *brachet1)))
     kn_cat ”story”))

(m359!
 (act (m358 (action (m210 (lex say that)))
       (object
        (m357 (act (m356 (action (m355 (lex take))) (object b14)))
         (agent b17)))))
 (agent b19) (kn_cat story))

(m359!)

 CPU time : 0.08

*
;; Newly inferred information:
;;
;; None.

;; In the story, the lady says that the brachet belongs to her.
(describe
(add agent *lady1
     act (build action (build lex ”say that”)
                object (build possessor *lady1
                              object *brachet1
                              rel (build lex ”brachet”)))
     kn_cat ”story”))

(m362!
 (act (m361 (action (m210 (lex say that)))
       (object
        (m360 (object b14) (possessor b19) (rel (m250 (lex brachet)))))))
 (agent b19) (kn_cat story))
```

```
(m362!)

 CPU time : 0.04

*
;; Newly inferred information:
;;
;; None.

;; In the story, the lady says that she wants the brachet.
(describe
(add agent *lady1
     act (build action (build lex "say that")
                object (build agent *lady1
                              act (build action (build lex "want")
                                        object *brachet1)))
     kn_cat "story"))

(m368! (object b14) (property (m208 (lex valuable))))
(m367!
 (act (m365 (action (m210 (lex say that)))
       (object
        (m364! (act (m363 (action (m207 (lex want))) (object b14)))
         (agent b19)))))
 (agent b19))
(m366! (act (m365)) (agent b19) (kn_cat story))

(m368! m367! m366! m364!)

 CPU time : 0.06

*
;; Newly inferred information:
;;
;; The brachet is valuable.

;; What does brachet mean?
^
-->(defineNoun "brachet")
 Definition of brachet:
 Class Inclusions: animal,
 Possible Actions: bite buttock,
 Possible Properties: valuable, small, white,
 Possibly Similar Items: mammal, pony,
nil
 CPU time : 8.68

*

;;; This is the content of Karen's original "br.txt2" file  ;;;

;"'Then,' said Merlin, 'call sir Gawain, for he must bring back the white
;hart.  Also, sir, ye must call Sir Tor, for he must bring back the brachet
;and the knight, or else slay him. ...'"  [p67]

;; In the story, Merlin (from background) says that Sir Gawain (from BG) must bring
;;   the hart to the hall
```

```
(describe
(add     agent *Mer
         act (build action (build lex "say that")
                   object (build mode (build lex "must")
                                 object (build agent *SG
                                               act (build action (build lex "bring")
                                                         object *hart1)
                                               to *hall1)))
         kn_cat "story"))

(m375!
 (act (m374 (action (m210 (lex say that)))
       (object
        (m373 (mode (m369 (lex must)))
         (object
          (m372 (act (m371 (action (m370 (lex bring))) (object b12)))
           (agent b8) (to b13)))))))
 (agent b5) (kn_cat story))

(m375!)

 CPU time : 0.07

*
;; Newly inferred information:
;;
;; None.

;; In the story, Merlin says that an unknown thing must bring the
;;    brachet to the hall
(describe
(add     agent *Mer
         act (build action (build lex "say that")
                   object (build mode (build lex "must")
                                 object (build agent #SirTor
                                               act (build action (build lex "bring")
                                                         object *brachet1)
                                               to *hall1)))
         kn_cat "story"))

(m380!
 (act (m379 (action (m210 (lex say that)))
       (object
        (m378 (mode (m369 (lex must)))
         (object
          (m377 (act (m376 (action (m370 (lex bring))) (object b14)))
           (agent b20) (to b13)))))))
 (agent b5) (kn_cat story))

(m380!)

 CPU time : 0.07

*
;; Newly inferred information:
;;
;; None.
```

```
;; In the story, the thing that must bring the brachet to the hall is
;;   named Sir Tor
(describe
(assert object *SirTor proper-name (build lex "Sir Tor") kn_cat "story"))

(m382! (kn_cat story) (object b20) (proper-name (m381 (lex Sir Tor))))

(m382!)

 CPU time : 0.01

*
;; Newly inferred information:
;;
;; None.

;; In the story, Sir Tor is a knight
(describe
(assert member *SirTor class (build lex "knight") kn_cat "story-comp"))

(m383! (class (m44 (lex knight))) (kn_cat story-comp) (member b20))

(m383!)

 CPU time : 0.01

*
;; Newly inferred information:
;;
;; None.

;; In the story, Merlin says that Sir Tor must either bring the knight
;;  who took the brachet to the hall or he must slay the knight who
;;  took the brachet.
(describe
(add agent *Mer
     act (build action (build lex "say that")
                object (build mode (build lex "must")
                               object (build min 1 max 1
                                             arg (build agent *SirTor
                                                        act (build action (build lex "bring")
                                                                   object *knight1)
                                                        to *hall1)
                                             arg (build agent *SirTor
                                                        act (build action (build lex "slay")
                                                                   object *knight1)))))
     kn_cat "story"))

(m391!
 (act (m390 (action (m210 (lex say that)))
       (object
        (m389 (mode (m369 (lex must)))
         (object
          (m388 (min 1) (max 1)
           (arg
            (m387 (act (m386 (action (m106 (lex slay))) (object b17)))
             (agent b20))
            (m385 (act (m384 (action (m370 (lex bring))) (object b17)))
```

```
            (agent b20) (to b13)))))))))
 (agent b5) (kn_cat story))

(m391!)

 CPU time : 0.08

*
;; Newly inferred information:
;;
;; None.

;; What does brachet mean?
^
-->(defineNoun "brachet")
 Definition of brachet:
 Class Inclusions: animal,
 Possible Actions: bite buttock,
 Possible Properties: valuable, small, white,
 Possibly Similar Items: mammal, pony,
nil
 CPU time : 10.32

*
;;; This is the content of Karen's original "br.txt3" file  ;;;

;'Sir,' said the elder 'there came a white hart this way this day and
;many hounds chased him, and a white brachet was always next to him.'
;[p67]

;; In the story, there is an elder
(describe
(assert member #elder class (build lex "elder") kn_cat "story"))

(m392! (class (m217 (lex elder))) (kn_cat story) (member b21))

(m392!)

 CPU time : 0.00

*
;; Newly inferred information:
;;
;; None.

;; In the story, there is a hart
(describe
(assert member #hart2 class (build lex "hart") kn_cat "story"))

(m393! (class (m20 (lex hart))) (kn_cat story) (member b22))

(m393!)

 CPU time : 0.02

*
;; Newly inferred information:
;;
```

```
;; None.

;; In the story, the hart is white
(describe
(assert object *hart2 property (build lex "white") kn_cat "story"))

(m394! (kn_cat story) (object b22) (property (m88 (lex white))))

(m394!)

 CPU time : 0.01

*
;; Newly inferred information:
;;
;; None.

;; In the story, the elder says that the hart came to something.
(describe
(add    agent *elder
        act (build action (build lex "say that")
                  object (build agent *hart2
                                act (build action (build lex "come")
                                          place #elderplace)))
        kn_cat "story"))

(m399!
 (act (m398 (action (m210 (lex say that)))
       (object
        (m397 (act (m396 (action (m395 (lex come))) (place b23)))
         (agent b22)))))
 (agent b21) (kn_cat story))

(m399!)

 CPU time : 0.08

*
;; Newly inferred information:
;;
;; None.

;; In the story, the elder says that something is chasing the hart.
(describe
(add    agent *elder
        act (build action (build lex "say that")
                  object (build agent #hounds2
                                act (build action (build lex "chase")
                                          object *hart2)))
        kn_cat "story"))

(m403!
 (act (m402 (action (m210 (lex say that)))
       (object
        (m401 (act (m400 (action (m220 (lex chase))) (object b22)))
         (agent b24)))))
 (agent b21) (kn_cat story))
```

```
(m403!)

 CPU time : 0.09

*
;; Newly inferred information:
;;
;; None.

;; In the story, the thing chasing the hart is a hound
(describe
(assert member *hounds2  class (build lex "hound")
        kn_cat "story"))

(m404! (class (m34 (lex hound))) (kn_cat story) (member b24))

(m404!)

 CPU time : 0.00

*
;; Newly inferred information:
;;
;; None.

;; In the story, the elder says that something is next to the hart
(describe
(add    agent *elder
        act (build action (build lex "say that")
                  object (build object #brachet2
                                location (build sp-rel (build lex "next to")
                                                object *hart2)))
        kn_cat "story"))

(m408!
 (act (m407 (action (m210 (lex say that)))
       (object
        (m406 (location (m405 (object b22) (sp-rel (m253 (lex next to)))))
         (object b25)))))
 (agent b21) (kn_cat story))

(m408!)

 CPU time : 0.08

*
;; Newly inferred information:
;;
;; None.

;; In the story, the thing that is next to the hart is a brachet
(describe
(assert member *brachet2  class (build lex "brachet")
        kn_cat "story"))

(m409! (class (m250 (lex brachet))) (kn_cat story) (member b25))

(m409!)
```

```
 CPU time : 0.00

*
;; Newly inferred information:
;;
;; None.

;; In the story, the brachet is white
(describe
(assert object *brachet2 property (build lex "white") kn_cat "story"))

(m410! (kn_cat story) (object b25) (property (m88 (lex white))))

(m410!)

 CPU time : 0.01

*
;; Newly inferred information:
;;
;; None.

;; In the story, the two brachets are actually the same brachet.
(describe
(assert equiv *brachet1 equiv *brachet2 kn_cat "story-comp"))

(m411! (equiv b25 b14) (kn_cat story-comp))

(m411!)

 CPU time : 0.00

*
;; Newly inferred information:
;;
;; None.

;; In the story, the two harts are actually the same hart.
(describe
(assert equiv *hart1 equiv *hart2 kn_cat "story-comp"))

(m412! (equiv b22 b12) (kn_cat story-comp))

(m412!)

 CPU time : 0.01

*
;; Newly inferred information:
;;
;; None.

;; In the story, the two hounds are actually the same hound.
(describe
(assert equiv *hounds1 equiv *hounds2 kn_cat "story-comp"))

(m413! (equiv b24 b15) (kn_cat story-comp))
```

```
(m413!)

 CPU time : 0.01

*
;; Newly inferred information:
;;
;; None.

;; What does brachet mean?
^
-->(defineNoun "brachet")
 Definition of brachet:
 Class Inclusions: animal,
 Possible Actions: bite buttock,
 Possible Properties: valuable, small, white,
 Possibly Similar Items: mammal, pony,
nil
 CPU time : 13.44

*
;;; This is the content of Karen's original "br.txt4" file  ;;;

;When Sir Tor was ready, he mounted upon his horse's back, and rode after the
;knight with the brachet.  [71]

;; In the story, Sir Tor mounts something.
(describe
(add agent *SirTor
     act (build action (build lex "mount")
                object #torhorse kn_cat "story")))

(m422! (act (m421 (action (m335 (lex mount))) (kn_cat story) (object b26)))
 (agent b20))

(m422!)

 CPU time : 0.07

*
;; Newly inferred information:
;;
;; None.

;; In the story, the thing that Sir Tor mounts is his horse.
(describe
(assert object *torhorse rel (build lex "horse") possessor *SirTor
        kn_cat "story"))

(m423! (kn_cat story) (object b26) (possessor b20) (rel (m22 (lex horse))))

(m423!)

 CPU time : 0.01

*
;; Newly inferred information:
```

```
;;
;; None.

;; In the story, Sir Tor's horse is a horse.
(describe
(assert member *torhorse class (build lex "horse") kn_cat "story"))

(m424! (class (m22 (lex horse))) (kn_cat story) (member b26))

(m424!)

 CPU time : 0.00

*
;; Newly inferred information:
;;
;; None.

;; In the story, Sir Tor rides his horse
(describe
(add agent *SirTor
     act (build action (build lex "ride")
                object *torhorse)
        direction *knight1 kn_cat "story"))

(m426! (act (m425 (action (m345 (lex ride))) (object b26))) (agent b20)
 (direction b17) (kn_cat story))

(m426!)

 CPU time : 0.06

*
;; Newly inferred information:
;;
;; None.

;; In the story, the knight who took the brachet is with the brachet.
(describe
(assert object1 *knight1 rel (build lex "with") object2 *brachet1
        kn_cat "story"))

(m428! (kn_cat story) (object1 b17) (object2 b14) (rel (m427 (lex with))))

(m428!)

 CPU time : 0.00

*
;; Newly inferred information:
;;
;; None.

;
;'Ye shall say [you are sent]  by the knight who went in quest of the knight
;with the brachet.'  [71]

;; In the story, Sir Tor says that something shall say that it was sent by the knight
```

```
;;  who is questing for the brachet.
(describe
(add agent *SirTor
     act (build action (build lex "say that")
                object (build mode (build lex "shall")
                              object (build agent #guys
                                            act (build action (build lex "say that")
                                                       object (build
                                                                agent (build skf "questor-for"
                                                                a1 *knight1
                                                                a2 *brachet1)
                                                                act (build object *guys
                                                                          action (build
                                                                            lex "send")))))))
        kn_cat "story"))

(m438!
 (act (m437 (action (m210 (lex say that)))
       (object
        (m436 (mode (m429 (lex shall)))
         (object
          (m435
           (act (m434 (action (m210))
                 (object
                  (m433 (act (m432 (action (m431 (lex send))) (object b27)))
                   (agent (m430 (a1 b17) (a2 b14) (skf questor-for)))))))
            (agent b27)))))))
 (agent b20) (kn_cat story))

(m438!)

 CPU time : 0.10

*
;; Newly inferred information:
;;
;; None.

;; In the story, Sir Tor is the knight who is questing for the brachet
(describe
(assert equiv (build skf "questor-for" a1 *knight1 a2 *brachet1)
        equiv *SirTor  kn_cat "story-comp"))

(m439! (equiv (m430 (a1 b17) (a2 b14) (skf questor-for)) b20)
 (kn_cat story-comp))

(m439!)

 CPU time : 0.02

*
;; Newly inferred information:
;;
;; None.

;; What does brachet mean?
^
-->(defineNoun "brachet")
```

```
 Definition of brachet:
 Class Inclusions: animal,
 Possible Actions: bite buttock,
 Possible Properties: valuable, small, white,
 Possibly Similar Items: mammal, pony,
nil
 CPU time : 14.85

*
;
;`I know you ride after the knight with the white brachet, and I shall bring
; you where he is,' said the dwarf.  [72]

; In the story, there is a dwarf
(describe
(assert member #dwarf1  class (build lex "dwarf") kn_cat "story"))

(m441! (class (m440 (lex dwarf))) (kn_cat story) (member b28))

(m441!)

 CPU time : 0.01

*
;; Newly inferred information:
;;
;; None.

;; In the story, the dwarf says that he knows that Sir Tor seeks the knight who took the brachet.
(describe
(add agent *dwarf1
     act (build action (build lex "say that")
                object (build agent *dwarf1
                              act (build action (build lex "know that")
                                        object (build agent *SirTor
                                                      act (build action (build lex "seek")
                                                                object *knight1)))))
     kn_cat "story"))

(m449!
 (act (m448 (action (m210 (lex say that)))
       (object
        (m447
         (act (m446 (action (m442 (lex know that)))
               (object
                (m445 (act (m444 (action (m443 (lex seek))) (object b17)))
                 (agent b20)))))
         (agent b28)))))
 (agent b28) (kn_cat story))

(m449!)

 CPU time : 0.10

*
;; Newly inferred information:
;;
;; None.
```

```
;; In the story, the dwarf says that he will bring Sir Tor someplace.
(describe
(add agent *dwarf1
     act (build action (build lex "say that")
                object (build agent *dwarf1
                              act (build action (build lex "bring")
                              object *SirTor to #brplace)))
     kn_cat "story"))

(m453!
 (act (m452 (action (m210 (lex say that)))
       (object
        (m451 (act (m450 (action (m370 (lex bring))) (object b20) (to b29)))
         (agent b28)))))
 (agent b28) (kn_cat story))

(m453!)

 CPU time : 0.09

*
;; Newly inferred information:
;;
;; None.

;; In the story, the knight is in the place that the dwarf is taking Sir Tor.
(describe
(assert object *knight1 location *brplace kn_cat "story"))

(m454! (kn_cat story) (location b29) (object b17))

(m454!)

 CPU time : 0.06

*
;; Newly inferred information:
;;
;; None.

;; In the story, the brachet is in the place that the dwarf is taking Sir Tor.
(describe
(assert object *brachet1 location *brplace kn_cat "story-comp"))

(m455! (kn_cat story-comp) (location b29) (object b14))

(m455!)

 CPU time : 0.01

*
;; Newly inferred information:
;;
;; None.

;;; This is the content of Karen's original "br.txt5" file  ;;;
```

```
;Then he went to the other pavilion and found a lady lying sleeping therein;
;and there was the white brachet which bayed at him fast.

;; In the story, Sir Tor goes to some place
(describe
(add agent *SirTor act (build action (build lex "go")) to #pavilion1 kn_cat "story"))

(m458! (act (m457 (action (m456 (lex go)))))) (agent b20) (kn_cat story)
 (to b30))

(m458!)

 CPU time : 0.09

*
;; Newly inferred information:
;;
;; None.

;; In the story, the place that Sir Tor goes to is a pavilion
(describe
(assert member *pavilion1  class (build lex "pavilion")
        kn_cat "story"))

(m460! (class (m459 (lex pavilion))) (kn_cat story) (member b30))

(m460!)

 CPU time : 0.01

*
;; Newly inferred information:
;;
;; None.

;; In the story, the lady is sleeping in the pavilion
(describe
(add agent *lady1 act (build action (build lex "sleep") place *pavilion1) kn_cat "story"))

(m463! (act (m462 (action (m461 (lex sleep))) (place b30))) (agent b19)
 (kn_cat story))

(m463!)

 CPU time : 0.08

*
;; Newly inferred information:
;;
;; None.

;; In the story, Sir Tor finds the lady in the pavilion
(describe
(add agent *SirTor
     act (build action (build lex "find")
               object *lady1)
     place *pavilion1
     kn_cat "story"))
```

```
(m466! (act (m465 (action (m464 (lex find))) (object b19))) (agent b20)
 (kn_cat story) (place b30))

(m466!)

 CPU time : 0.10

*
;; Newly inferred information:
;;
;; None.

;; In the story, Sir Tor finds the brachet in the pavilion
(describe
(add agent *SirTor
     act (build action (build lex "find")
                object *brachet1)
     place *pavilion1
     kn_cat "story"))

(m468! (act (m467 (action (m464 (lex find))) (object b14))) (agent b20)
 (kn_cat story) (place b30))

(m468!)

 CPU time : 0.09

*
;; Newly inferred information:
;;
;; None.

;; In the story, the brachet bays at Sir Tor.
(describe
(add agent *brachet1
     act (build action (build lex "bay")
                direction *SirTor)
     kn_cat "story"))

(m470! (act (m469 (action (m111 (lex bay))) (direction b20))) (agent b14)
 (kn_cat story))
(m289! (act (m112 (action (m111)))) (agent b14))
(m259! (subclass (m250 (lex brachet))) (superclass (m34 (lex hound))))

(m470! m289! m259!)

 CPU time : 0.22

*
;; Newly inferred information:
;;
;; The brachet bays.
;; Brachet is a subclass of hound.

;; What does brachet mean?
^
--> (defineNoun "brachet")
```

```
 Definition of brachet:
 Class Inclusions: hound, dog,
 Possible Actions: bite buttock, bay, hunt,
 Possible Properties: valuable, small, white,
nil
 CPU time : 84.34

*
;;; This is the content of Karen's original "br.txt6" file  ;;;

;As soon as Sir Tor spied the white brachet, he took it by force and gave it
;to the dwarf.

;; In the story, Sir Tor spys the brachet
(describe
(add agent *SirTor
     act (build action (build lex "spy")
                object *brachet1)
     kn_cat "story"))

(m500! (act (m499 (action (m498 (lex spy))) (object b14))) (agent b20)
 (kn_cat story))

(m500!)

 CPU time : 0.10

*
;; Newly inferred information:
;;
;; None.

;; In the story, Sir Tor takes the bracket
(describe
(add agent *SirTor
     act (build action (build lex "take")
                object *brachet1)
     kn_cat "story"))

(m501! (act (m356 (action (m355 (lex take))) (object b14))) (agent b20)
 (kn_cat story))

(m501!)

 CPU time : 0.14

*
;; Newly inferred information:
;;
;; None.

;; In the story, Sir Tor gives the brachet to the dwarf
(describe
(add agent *SirTor
     act (build action (build lex "give")
                object *brachet1
                indobj *dwarf1)
        kn_cat "story"))
```

```
(m504! (act (m503 (action (m502 (lex give))) (indobj b28) (object b14)))
 (agent b20) (kn_cat story))

(m504!)

 CPU time : 0.07

*
;; Newly inferred information:
;;
;; None.

;
;With the noise, the lady came out of the pavilion with all her damosels.

;; In the story, the lady comes from the pavilion
(describe
(add agent *lady1 act (build action (build lex "come") from *pavilion1) kn_cat "story"))

(m506! (act (m505 (action (m395 (lex come))) (from b30))) (agent b19)
 (kn_cat story))

(m506!)

 CPU time : 0.10

*
;; Newly inferred information:
;;
;; None.

;; In the story, there is a noise
(describe
(assert member #noise1  class (build lex noise) kn_cat "story"))

(m508! (class (m507 (lex noise))) (kn_cat story) (member b31))

(m508!)

 CPU time : 0.01

*
;; Newly inferred information:
;;
;; None.

;; In the story, the noise caused the lady to come from the pavilion.
(describe
(add    cause *noise1
        effect (build agent *lady1 act (build action (build lex "come") from *pavilion1))
        kn_cat "story"))

(m510! (cause b31)
 (effect
  (m509 (act (m505 (action (m395 (lex come))) (from b30))) (agent b19)))
 (kn_cat story))
```

```
(m510!)

 CPU time : 0.09

*
;; Newly inferred information:
;;
;; None.

;; In the story, there is a damosel
(describe
(assert member *damosels1  class (build lex "damosel")
        kn_cat "story"))

(m512! (class (m511 (lex damosel))) (kn_cat story))

(m512!)

 CPU time : 0.03

*
;; In the story, the damosel comes from the pavilion
(describe
(add agent #damosels1 act (build action (build lex "come") from *pavilion1) kn_cat "story"))

(m513! (act (m505 (action (m395 (lex come))) (from b30))) (agent b32)
 (kn_cat story))

(m513!)

 CPU time : 0.10

*
;; Newly inferred information:
;;
;; None.

;; In the story, the noise caused the damosel to come from the pavilion
(describe
(add    cause *noise1
        effect (build agent *damosels1 act (build action
                                             (build lex "come") from *pavilion1))
        kn_cat "story"))

(m515! (cause b31)
 (effect
  (m514 (act (m505 (action (m395 (lex come))) (from b30))) (agent b32)))
 (kn_cat story))

(m515!)

 CPU time : 0.09

*
;; In the story, the damosel is a person
(describe
(assert member *damosels1 class (build lex "person") kn_cat "story-comp"))
```

```
(m516! (class (m38 (lex person))) (kn_cat story-comp) (member b32))

(m516!)

 CPU time : 0.01

*
;; Newly inferred information:
;;
;; None.

;; In the story, the noise was the brachet baying.
(describe
(assert equiv *noise1 equiv (build agent *brachet1 act (build action (build lex "bay")))
        kn_cat "story_comp"))

(m517! (equiv b31 (m289! (act (m112 (action (m111 (lex bay))))) (agent b14)))
 (kn_cat story_comp))

(m517!)

 CPU time : 0.01

*
;; Newly inferred information:
;;
;; None.

;;; 'What!  Will ye take my brachet from me?' said the lady.

;; In the story, the lady asks Sir Tor why he took the brachet from her.
(describe
(add agent *lady1
     act (build action (build lex "ask")
               object (build agent *SirTor
                             act (build action (build lex "take")
                                       object *brachet1)
                             from *lady1))
               indobj *SirTor kn_cat "story"))

(m521!
 (act (m520 (action (m518 (lex ask)))
       (object
        (m519 (act (m356 (action (m355 (lex take))) (object b14)))
         (agent b20) (from b19)))))
 (agent b19) (indobj b20) (kn_cat story))

(m521!)

 CPU time : 0.10

*
;; Newly inferred information:
;;
;; None.

;; In the story, the lady is angry.
(describe
```

```
(assert object *lady1 property (build lex "angry") kn_cat "story-comp"))

(m523! (kn_cat story-comp) (object b19) (property (m522 (lex angry))))

(m523!)

 CPU time : 0.01

*
;; Newly inferred information:
;;
;; None.

;; In the story,Sir Tor taking the brachet caused the lady to be angry.
(describe
(assert cause (build agent *SirTor
                     act (build action (build lex "take")
                                 object *brachet1)
                     from *lady1)
        effect (build object *lady1 property (build lex "angry"))
        kn_cat "story-comp"))

(m525!
 (cause
  (m519 (act (m356 (action (m355 (lex take))) (object b14))) (agent b20)
   (from b19)))
 (effect (m524 (object b19) (property (m522 (lex angry)))))
 (kn_cat story-comp))

(m525!)

 CPU time : 0.01

*
;; Newly inferred information:
;;
;; None.

;
; 'Yes,' said Sir Tor, 'this brachet I have sought from King Arthur's court
; hither.'  [72]

;; In the story,Sir Tor tells the lady that he took the brachet from her.
(describe
(add agent *SirTor
     act (build action (build lex "say that")
                object (build agent *SirTor
                              act (build action (build lex "take")
                                          object *brachet1)
                              from *lady1)
                indobj *lady1)  kn_cat "story"))

(m527!
 (act (m526 (action (m210 (lex say that))) (indobj b19)
       (object
        (m519 (act (m356 (action (m355 (lex take))) (object b14)))
         (agent b20) (from b19)))))
 (agent b20) (kn_cat story))
```

```
(m527!)

 CPU time : 0.10

*
;; Newly inferred information:
;;
;; None.

;; In the story,Sir Tor says that he sought the brachet from the hall to the pavilion.
(describe
(add agent *SirTor
     act (build action (build lex "say that")
                object (build agent *SirTor
                              act (build action (build lex "seek")
                                        object *brachet1)
                              from *hall1
                              to *pavilion1))
     kn_cat "story"))

(m531!
 (act (m530 (action (m210 (lex say that)))
       (object
        (m529 (act (m528 (action (m443 (lex seek))) (object b14)))
         (agent b20) (from b13) (to b30)))))
 (agent b20) (kn_cat story))

(m531!)

 CPU time : 0.10

*
;; Newly inferred information:
;;
;; None.

;; What does brachet mean?
^
-->(defineNoun "brachet")
 Definition of brachet:
 Class Inclusions: dog,
 Possible Actions: bite buttock, bay, hunt,
 Possible Properties: valuable, small, white,
nil
 CPU time : 24.39

*

End of /projects/stn2/CVA/demos/brachet.demo demonstration.

 CPU time : 193.37

*
```

B.2 Cat

```
Starting image '/util/acl62/composer'
```

```
  with no arguments
  in directory '/projects/stn2/CVA/'
  on machine 'localhost'.

International Allegro CL Enterprise Edition
6.2 [Solaris] (Aug 15, 2002 14:24)
Copyright (C) 1985-2002, Franz Inc., Berkeley, CA, USA.  All Rights Reserved.

This development copy of Allegro CL is licensed to:
   [4549] SUNY/Buffalo, N. Campus

;; Optimization settings: safety 1, space 1, speed 1, debug 2.
;; For a complete description of all compiler switches given the current
;; optimization settings evaluate (explain-compiler-settings).
;;---
;; Current reader case mode: :case-sensitive-lower
cl-user(1): :ld /projects/snwiz/bin/sneps
; Loading /projects/snwiz/bin/sneps.lisp
Loading system SNePS...10% 20% 30% 40% 50% 60% 70% 80% 90% 100%
SNePS-2.6 [PL:0a 2002/09/30 22:37:46] loaded.
Type '(sneps)' or '(snepslog)' to get started.
cl-user(2): :ld defun_noun.cl
; Loading /projects/stn2/CVA/defun_noun.cl
cl-user(3): (sneps)

   Welcome to SNePS-2.6 [PL:0a 2002/09/30 22:37:46]

Copyright (C) 1984--2002 by Research Foundation of
State University of New York. SNePS comes with ABSOLUTELY NO WARRANTY!
Type '(copyright)' for detailed copyright information.
Type '(demo)' for a list of example applications.

   2/24/2003 15:49:35

* (demo "demos/cat.demo")

File /projects/stn2/CVA/demos/cat.demo is now the source of input.

 CPU time : 0.02

* ;;; Reset the network
(resetnet t)

Net reset

 CPU time : 0.03

*
;;; Don't trace infer
^(
--> setq snip:*infertrace* nil)
nil

 CPU time : 0.00

*
;;; Load all valid relations
(intext "/projects/stn2/CVA/demos/rels")
```

```
File /projects/stn2/CVA/demos/rels is now the source of input.

 CPU time : 0.00

*

(a1 a2 a3 a4 after agent antonym associated before cause class direction equiv
 etime from in indobj instr into lex location manner member mode object on
 onto part place possessor proper-name property rel skf sp-rel stime subclass
 superclass synonym time to whole kn_cat)

 CPU time : 0.22

*

End of file /projects/stn2/CVA/demos/rels

 CPU time : 0.23

*
;;; Compose paths
(intext "/projects/stn2/CVA/demos/paths")
File /projects/stn2/CVA/demos/paths is now the source of input.

 CPU time : 0.01

*
before implied by the path (compose before (kstar (compose after- ! before)))
before- implied by the path (compose (kstar (compose before- ! after)) before-)

 CPU time : 0.00

*
after implied by the path (compose after (kstar (compose before- ! after)))
after- implied by the path (compose (kstar (compose after- ! before)) after-)

 CPU time : 0.00

*
class implied by the path (compose class
                           (kstar (compose subclass- ! superclass)))
class- implied by the path (compose (kstar (compose superclass- ! subclass))
                            class-)

 CPU time : 0.00

*
subclass implied by the path (compose subclass
                              (kstar (compose superclass- ! subclass)))
subclass- implied by the path (compose
                               (kstar (compose subclass- ! superclass))
                               subclass-)

 CPU time : 0.00

*

End of file /projects/stn2/CVA/demos/paths
```

```
 CPU time : 0.03

*
;;; Load Background Knowledge
(demo "/projects/stn2/CVA/demos/cat.base")

File /projects/stn2/CVA/demos/cat.base is now the source of input.

 CPU time : 0.01

* ;;; Knowledge Base for initial cat demo

;;; Harts are deer
(describe
(assert subclass (build lex "hart")
        superclass (build lex "deer")))

(m3! (subclass (m1 (lex hart))) (superclass (m2 (lex deer))))

(m3!)

 CPU time : 0.01

*
;;; Halls are buildings
(describe
(assert subclass (build lex "hall")
        superclass (build lex "building")))

(m6! (subclass (m4 (lex hall))) (superclass (m5 (lex building))))

(m6!)

 CPU time : 0.01

*
;;; Hounds are dogs
(describe
(assert subclass (build lex "hound")
        superclass (build lex "dog")))

(m9! (subclass (m7 (lex hound))) (superclass (m8 (lex dog))))

(m9!)

 CPU time : 0.01

*
;;; Dogs are mammals
(describe
(assert subclass (build lex "dog")
        superclass (build lex "mammal")))

(m11! (subclass (m8 (lex dog))) (superclass (m10 (lex mammal))))

(m11!)
```

```
 CPU time : 0.01

*
;;; Dogs are quadrupeds
(describe
(assert subclass (build lex "dog")
        superclass (build lex "quadruped")))

(m13! (subclass (m8 (lex dog))) (superclass (m12 (lex quadruped))))

(m13!)

 CPU time : 0.01

*
;;; Dogs are carnivores
(describe
(assert subclass (build lex "dog")
        superclass (build lex "carnivore")))

(m15! (subclass (m8 (lex dog))) (superclass (m14 (lex carnivore))))

(m15!)

 CPU time : 0.00

*
;;; Dogs are predators
(describe
(assert subclass (build lex "dog")
        superclass (build lex "predator")))

(m17! (subclass (m8 (lex dog))) (superclass (m16 (lex predator))))

(m17!)

 CPU time : 0.01

*
;;; Dogs are animals
(describe
(assert subclass (build lex "dog")
        superclass (build lex "animal")))

(m19! (subclass (m8 (lex dog))) (superclass (m18 (lex animal))))

(m19!)

 CPU time : 0.01

*
;;; Rex is a dog
;;; (Ehrlich never had an object - proper-name case frame for Rex)

(describe
(assert member #Rex
        class (build lex "dog")))
```

```
(m20! (class (m8 (lex dog))) (member b1))

(m20!)

 CPU time : 0.00

*
(describe
(assert object *Rex
        proper-name (build lex "Rex")))

(m22! (object b1) (proper-name (m21 (lex Rex))))

(m22!)

 CPU time : 0.01

*
;;; Rex belongs to a person

(describe
(assert object *Rex
        possessor #person1
        rel (build lex "dog")))

(m23! (object b1) (possessor b2) (rel (m8 (lex dog))))

(m23!)

 CPU time : 0.01

*
(describe
(assert member *person1
        class (build lex "person")))

(m25! (class (m24 (lex person))) (member b2))

(m25!)

 CPU time : 0.01

*
;;; Deer are mammals
(describe
(assert subclass (build lex "deer")
        superclass (build lex "mammal")))

(m26! (subclass (m2 (lex deer))) (superclass (m10 (lex mammal))))

(m26!)

 CPU time : 0.00

*
;;; Deer are quadrupeds
(describe
(assert subclass (build lex "deer")
```

```
        superclass (build lex "quadruped")))

(m27! (subclass (m2 (lex deer))) (superclass (m12 (lex quadruped))))

(m27!)

 CPU time : 0.00

*
;;; Deer are herbivores
(describe
(assert subclass (build lex "deer")
        superclass (build lex "herbivore")
        ))

(m29! (subclass (m2 (lex deer))) (superclass (m28 (lex herbivore))))

(m29!)

 CPU time : 0.01

*
;;; Carnivores are animals
(describe
(assert subclass (build lex "carnivore")
        superclass (build lex "animal")
        ))

(m30! (subclass (m14 (lex carnivore))) (superclass (m18 (lex animal))))

(m30!)

 CPU time : 0.00

*
;;; Predators are carnivores
(describe
(assert subclass (build lex "predator")
        superclass (build lex "carnivore")
        ))

(m31! (subclass (m16 (lex predator))) (superclass (m14 (lex carnivore))))

(m31!)

 CPU time : 0.00

*
;;; "Herbivore" and "carnivore" are antonyms
(describe
(assert antonym (build lex "herbivore")
        antonym (build lex "carnivore")))

(m32! (antonym (m28 (lex herbivore)) (m14 (lex carnivore))))

(m32!)

 CPU time : 0.01
```

```
*
;;; "Predator" and "carnivore" are antonyms
(describe
(assert antonym (build lex "predator")
        antonym (build lex "carnivore")))

(m33! (antonym (m16 (lex predator)) (m14 (lex carnivore))))

(m33!)

 CPU time : 0.01

*
;;; Mammals are animals
(describe
(assert subclass (build lex "mammal")
        superclass (build lex "animal")
        ))

(m34! (subclass (m10 (lex mammal))) (superclass (m18 (lex animal))))

(m34!)

 CPU time : 0.00

*
;;; Mammals are vertebrates
(describe
(assert subclass (build lex "mammal")
        superclass (build lex "vertebrate")
        ))

(m36! (subclass (m10 (lex mammal))) (superclass (m35 (lex vertebrate))))

(m36!)

 CPU time : 0.01

*
;;; Quadrupeds are vertebrates
(describe
(assert subclass (build lex "quadruped")
        superclass (build lex "vertebrate")
        ))

(m37! (subclass (m12 (lex quadruped))) (superclass (m35 (lex vertebrate))))

(m37!)

 CPU time : 0.00

*
;;; Animals are physical objects
(describe
(assert subclass (build lex "animal")
        superclass (build lex "phys obj")
        ))
```

```
(m39! (subclass (m18 (lex animal))) (superclass (m38 (lex phys obj))))

(m39!)

 CPU time : 0.01

*
;;; The Round Table is a table
(describe
(assert member (build lex "Round Table")
        class (build lex "table")
        ))

(m42! (class (m41 (lex table))) (member (m40 (lex Round Table))))

(m42!)

 CPU time : 0.01

*
;;; King Arthur is a king
(describe
(assert object #KingArt
        proper-name (build lex "King Arthur")
        ))

(m44! (object b3) (proper-name (m43 (lex King Arthur))))

(m44!)

 CPU time : 0.01

*
(describe
(assert member *KingArt

        class (build lex "king")
        ))

(m46! (class (m45 (lex king))) (member b3))

(m46!)

 CPU time : 0.00

*
;;; The Round Table is King Arthur's table
(describe
(assert object (build lex "Round Table")
        possessor *KingArt
        rel (build lex "table")
        ))

(m47! (object (m40 (lex Round Table))) (possessor b3) (rel (m41 (lex table))))

(m47!)
```

```
 CPU time : 0.01

*
;;; Excalibur is a sword
(describe
(assert object #Excal
        proper-name (build lex "Excalibur")
        ))

(m49! (object b4) (proper-name (m48 (lex Excalibur))))

(m49!)

 CPU time : 0.01

*
(describe
(assert member *Excal
        class (build lex "sword")
        ))

(m51! (class (m50 (lex sword))) (member b4))

(m51!)

 CPU time : 0.00

*
;;; Excalibur is King Arthur's sword
(describe
(assert object *Excal
        possessor *KingArth
        rel (build lex "sword")
        ))

(m52! (object b4) (rel (m50 (lex sword))))

(m52!)

 CPU time : 0.00

*
;;; Merlin is a wizard
(describe
(assert object #Merlin
        proper-name (build lex "Merlin")
        ))

(m54! (object b5) (proper-name (m53 (lex Merlin))))

(m54!)

 CPU time : 0.01

*
(describe
(assert member *Merlin
```

```
        class (build lex "wizard")
        ))

(m56! (class (m55 (lex wizard))) (member b5))

(m56!)

 CPU time : 0.01

*
;;; Wizards are persons
(describe
(assert subclass (build lex "wizard")
        superclass (build lex "person")
        ))

(m57! (subclass (m55 (lex wizard))) (superclass (m24 (lex person))))

(m57!)

 CPU time : 0.01

*
;;; King Ban is a king
(describe
(assert object #KingBan
        proper-name (build lex "King Ban")
        ))

(m59! (object b6) (proper-name (m58 (lex King Ban))))

(m59!)

 CPU time : 0.02

*
(describe
(assert member *KingBan

        class (build lex "king")
        ))

(m60! (class (m45 (lex king))) (member b6))

(m60!)

 CPU time : 0.01

*
;;; King Bors is a king
(describe
(assert object #KingBors
        proper-name (build lex "King Bors")
        ))

(m62! (object b7) (proper-name (m61 (lex King Bors))))

(m62!)
```

```
 CPU time : 0.01

*
(describe
(assert member *KingBors

        class (build lex "king")
        ))

(m63! (class (m45 (lex king))) (member b7))

(m63!)

 CPU time : 0.00

*
;;; King Lot is a king
(describe
(assert object #KingLot
        proper-name (build lex "King Lot")
        ))

(m65! (object b8) (proper-name (m64 (lex King Lot))))

(m65!)

 CPU time : 0.01

*
(describe
(assert member *KingLot

        class (build lex "king")
        ))

(m66! (class (m45 (lex king))) (member b8))

(m66!)

 CPU time : 0.01

*
;;; Sir Galahad is a knight
(describe
(assert object #SirGal
        proper-name (build lex "Sir Galahad")
        ))

(m68! (object b9) (proper-name (m67 (lex Sir Galahad))))

(m68!)

 CPU time : 0.01

*
(describe
(assert member *SirGal
```

```
        class (build lex "knight")
        ))

(m70! (class (m69 (lex knight))) (member b9))

(m70!)

 CPU time : 0.00

*
;;; Sir Gawain is a knight
(describe
(assert object #SirGaw
        proper-name (build lex "Sir Gawain")
        ))

(m72! (object b10) (proper-name (m71 (lex Sir Gawain))))

(m72!)

 CPU time : 0.01

*
(describe
(assert member *SirGaw

        class (build lex "knight")
        ))

(m73! (class (m69 (lex knight))) (member b10))

(m73!)

 CPU time : 0.01

*
;;; Sir Tristram is a knight
(describe
(assert object #SirTris
        proper-name (build lex "Sir Tristram")
        ))

(m75! (object b11) (proper-name (m74 (lex Sir Tristram))))

(m75!)

 CPU time : 0.01

*
(describe
(assert member *SirTris

        class (build lex "knight")
        ))

(m76! (class (m69 (lex knight))) (member b11))
```

```
(m76!)

 CPU time : 0.00

*
;;; Sideboards are furniture
(describe
(assert subclass (build lex "sideboard")
        superclass (build lex "furniture")
        ))

(m79! (subclass (m77 (lex sideboard))) (superclass (m78 (lex furniture))))

(m79!)

 CPU time : 0.02

*
;;; Tables are furniture
(describe
(assert subclass (build lex "table")
        superclass (build lex "furniture")
        ))

(m80! (subclass (m41 (lex table))) (superclass (m78 (lex furniture))))

(m80!)

 CPU time : 0.01

*
;;; Chairs are furniture
(describe
(assert subclass (build lex "chair")
        superclass (build lex "furniture")
        ))

(m82! (subclass (m81 (lex chair))) (superclass (m78 (lex furniture))))

(m82!)

 CPU time : 0.01

*
;;; Horses are quadrupeds
(describe
(assert subclass (build lex "horse")
        superclass (build lex "quadruped")
        ))

(m84! (subclass (m83 (lex horse))) (superclass (m12 (lex quadruped))))

(m84!)

 CPU time : 0.01

*
;;; Horses are herbivores
```

```
(describe
(assert subclass (build lex "horse")
        superclass (build lex "herbivore")
        ))

(m85! (subclass (m83 (lex horse))) (superclass (m28 (lex herbivore))))

(m85!)

 CPU time : 0.01

*
;;; Ungulates are herbivores
(describe
(assert subclass (build lex "ungulate")
        superclass (build lex "herbivore")
        ))

(m87! (subclass (m86 (lex ungulate))) (superclass (m28 (lex herbivore))))

(m87!)

 CPU time : 0.03

*
;;; Horses are animals
(describe
(assert subclass (build lex "horse")
        superclass (build lex "animal")
        ))

(m88! (subclass (m83 (lex horse))) (superclass (m18 (lex animal))))

(m88!)

 CPU time : 0.01

*
;;; Knights are persons
(describe
(assert subclass (build lex "knight")
        superclass (build lex "person")
        ))

(m89! (subclass (m69 (lex knight))) (superclass (m24 (lex person))))

(m89!)

 CPU time : 0.01

*
;;; "Person" is a basic-level category
(describe
(assert member (build lex "person")

        class (build lex "basic ctgy")
        ))
```

```
(m91! (class (m90 (lex basic ctgy))) (member (m24 (lex person))))

(m91!)

 CPU time : 0.01

*
;;; "Dog" is a basic-level category
(describe
(assert member (build lex "dog")

        class (build lex "basic ctgy")
        ))

(m92! (class (m90 (lex basic ctgy))) (member (m8 (lex dog))))

(m92!)

 CPU time : 0.00

*
;;; "Horse" is a basic-level category
(describe
(assert member (build lex "horse")

        class (build lex "basic ctgy")
        ))

(m93! (class (m90 (lex basic ctgy))) (member (m83 (lex horse))))

(m93!)

 CPU time : 0.02

*
;;; "Deer" is a basic-level category
(describe
(assert member (build lex "deer")

        class (build lex "basic ctgy")
        ))

(m94! (class (m90 (lex basic ctgy))) (member (m2 (lex deer))))

(m94!)

 CPU time : 0.01

*
;;; "Chair" is a basic-level category
(describe
(assert member (build lex "chair")

        class (build lex "basic ctgy")
        ))

(m95! (class (m90 (lex basic ctgy))) (member (m81 (lex chair))))
```

```
(m95!)

 CPU time : 0.02

*
;;; "Table" is a basic-level category
(describe
(assert member (build lex "table")

        class (build lex "basic ctgy")
        ))

(m96! (class (m90 (lex basic ctgy))) (member (m41 (lex table))))

(m96!)

 CPU time : 0.00

*
;;; White is a color
(describe
(assert member (build lex "white")
        class (build lex "color")
        ))

(m99! (class (m98 (lex color))) (member (m97 (lex white))))

(m99!)

 CPU time : 0.02

*
;;; Black is a color
(describe
(assert member (build lex "black")
        class (build lex "color")
        ))

(m101! (class (m98 (lex color))) (member (m100 (lex black))))

(m101!)

 CPU time : 0.01

*
;;; Small is a size
(describe
(assert member (build lex "small")
        class (build lex "size")
        ))

(m104! (class (m103 (lex size))) (member (m102 (lex small))))

(m104!)

 CPU time : 0.01

*
```

```
;;; "Small" and "little" are synonyms
(describe
(assert synonym (build lex "small")
        synonym (build lex "little")
        ))

(m106! (synonym (m105 (lex little)) (m102 (lex small))))

(m106!)

 CPU time : 0.01

*
;;; Large is a size
(describe
(assert member (build lex "large")
        class (build lex "size")
        ))

(m108! (class (m103 (lex size))) (member (m107 (lex large))))

(m108!)

 CPU time : 0.02

*
;;; "Large" and "big" are synonyms
(describe
(assert synonym (build lex "large")
        synonym (build lex "big")
        ))

(m110! (synonym (m109 (lex big)) (m107 (lex large))))

(m110!)

 CPU time : 0.02

*
;;; Good is a value
(describe
(assert member (build lex "good")
        class (build lex "value")
        ))

(m113! (class (m112 (lex value))) (member (m111 (lex good))))

(m113!)

 CPU time : 0.01

*
;;; Bad is a value
(describe
(assert member (build lex "bad")
        class (build lex "value")
        ))
```

```
(m115! (class (m112 (lex value))) (member (m114 (lex bad))))

(m115!)

 CPU time : 0.01

*
;;; If x is a horse, then presumably there is a person, y, such that the
;;;    function of x is to be ridden by y
(describe
(assert forall $v2
        ant (build member *v2
                   class (build lex "horse"))
        cq ((build mode (build lex "presumably")
                   object (build object1 *v2
                                 class (build agent (build a2 *v2
                                                           skf "rider-of")
                                                    act (build action (build lex "ride")
                                                               object *v2))
                                 rel (build lex "function")))
            (build mode (build lex "presumably")
                   object (build member (build a2 *v2
                                                skf "rider-of")
                                 class (build lex "person"))))))

(m119! (forall v1) (ant (p1 (class (m83 (lex horse))) (member v1)))
 (cq
  (p8 (mode (m116 (lex presumably)))
   (object
    (p7 (class (m24 (lex person))) (member (p2 (a2 v1) (skf rider-of))))))
  (p6 (mode (m116))
   (object
    (p5
     (class
      (p4 (act (p3 (action (m117 (lex ride))) (object v1))) (agent (p2))))
     (object1 v1) (rel (m118 (lex function)))))))))

(m119!)

 CPU time : 0.04

*
;;; If a rider rides an animal that belongs to some class, then that class is
;;;    a subclass of equine

(describe
(assert forall (*v2 $v3)
        &ant ((build member *v2
                     class (build lex "animal"))
              (build member *v2
                     class *v3
                     )
              (build member (build a2 *v2
                                   skf "rider-of")
                     class (build lex "person"))
              (build agent (build a2 *v2
                                   skf "rider-of")
                     act (build action (build lex "ride")
```

```
                                object *v2)))
        cq (build subclass *v3
                  superclass (build lex "equine"))))

(m121! (forall v2 v1)
 (&ant (p10 (class v2) (member v1))
  (p9 (class (m18 (lex animal))) (member v1))
  (p7 (class (m24 (lex person))) (member (p2 (a2 v1) (skf rider-of))))
  (p4 (act (p3 (action (m117 (lex ride))) (object v1))) (agent (p2))))
 (cq (p11 (subclass v2) (superclass (m120 (lex equine))))))

(m121!)

 CPU time : 0.03

*

;; All animals who bay are hounds
(describe
 (assert forall ($hound1 $categ)
   &ant ((build agent *hound1 act (build action (build lex "bay")))
         (build member *hound1 class *categ)
         (build subclass *categ superclass (build lex "animal")))
   cq (build subclass *categ superclass (build lex "hound"))))

(m124! (forall v4 v3)
 (&ant (p14 (subclass v4) (superclass (m18 (lex animal))))
  (p13 (class v4) (member v3))
  (p12 (act (m123 (action (m122 (lex bay))))) (agent v3)))
 (cq (p15 (subclass v4) (superclass (m7 (lex hound))))))

(m124!)

 CPU time : 0.03

*
;;; If something bites, then it's an animal

(describe
;if it bites, it's an animal
(assert forall ($animal1 $bitten *categ)
        &ant ((build agent *animal1 act (build action (build lex "bite")
                                                object *bitten))
              (build member *animal1 class *categ))
        cq (build subclass *categ superclass (build lex "animal"))
        kn_cat "life-rule.1"))

(m126! (forall v6 v5 v4)
 (&ant (p18 (class v4) (member v5))
  (p17 (act (p16 (action (m125 (lex bite))) (object v6))) (agent v5)))
 (cq (p14 (subclass v4) (superclass (m18 (lex animal))))) (kn_cat life-rule.1))

(m126!)

 CPU time : 0.03

*
;;; If something sleeps, then it's an animal
```

```
(describe
(assert forall $v5
        ant (build act (build action (build lex "sleep"))
                    agent *v5)
        cq (build member *v5
                  class (build lex "animal"))
        ))

(m129! (forall v7)
 (ant (p19 (act (m128 (action (m127 (lex sleep))))) (agent v7)))
 (cq (p20 (class (m18 (lex animal))) (member v7))))

(m129!)

 CPU time : 0.03

*
;;; If something ambles, then it's an animal

(describe
(assert forall *v5
        ant (build act (build action (build lex "ambling"))
                    agent *v5)
        cq (build member *v5
                  class (build lex "animal"))
        ))

(m132! (forall v7)
 (ant (p21 (act (m131 (action (m130 (lex ambling))))) (agent v7)))
 (cq (p20 (class (m18 (lex animal))) (member v7))))

(m132!)

 CPU time : 0.02

*
;;; If something is an animal and a member of another class, then that class
;;;    is a subclass of animal

(describe
(assert forall (*v5 $v6)
        &ant ((build member *v5
                     class (build lex "animal"))
              (build member *v5
                     class *v6))
        cq (build subclass *v6
                  superclass (build lex "animal"))))

(m133! (forall v8 v7)
 (&ant (p22 (class v8) (member v7))
  (p20 (class (m18 (lex animal))) (member v7)))
 (cq (p23 (subclass v8) (superclass (m18)))))

(m133!)

 CPU time : 0.01
```

```
*
;;; If something is presumed to be an animal and a member of another class,
;;;    then presumably that class is a subclass of animal

(describe
(assert forall (*v5 *v6)
       &ant ((build mode (build lex "presumably")
                    object (build member *v5
                                 class (build lex "animal")))
             (build member *v5
                    class *v6))
       cq (build mode (build lex "presumably")
                 object (build subclass *v6
                               superclass (build lex "animal")))
       ))

(m134! (forall v8 v7)
 (&ant
  (p24 (mode (m116 (lex presumably)))
   (object (p20 (class (m18 (lex animal))) (member v7))))
  (p22 (class v8) (member v7)))
 (cq (p25 (mode (m116)) (object (p23 (subclass v8) (superclass (m18)))))))

(m134!)

CPU time : 0.02

*
;;; If something is a mammal and a member of another (non-superordinate) class,
;;;    then that class is a subclass of mammal

;;;(describe
;;;(assert forall (*v5 *v6)
;;;     &ant ((build member *v5
;;;                 class *v6)
;;;           (build member *v5
;;;                  class (build lex "mammal")
;;;                  ))
;;;     cq ((build subclass *v6
;;;                superclass (build lex "animal"))
;;;         (build subclass *v6
;;;                superclass (build lex "mammal")))
;;;     ))

;;; If someone belongs to a subclass of person then that someone is a person
(describe
(assert forall ($v8 $v9)
       &ant ((build subclass *v8
                    superclass (build lex "person"))
             (build member *v9
                    class *v8))
       cq (build member *v9
                 class (build lex "person"))
))

(m135! (forall v10 v9)
 (&ant (p27 (class v9) (member v10))
  (p26 (subclass v9) (superclass (m24 (lex person))))))
```

```
 (cq (p28 (class (m24)) (member v10))))

(m135!)

 CPU time : 0.01

*

;;; If a person can carry an object, then that object is small

(describe
(assert forall ($v13 *v9)
        &ant ((build member *v9
                     class (build lex "person"))
              (build act (build action (build lex "carry") object *v13)
                     agent *v9))
        cq (build object *v13
                  property (build lex "small"))
        ))

(m137! (forall v11 v10)
 (&ant (p30 (act (p29 (action (m136 (lex carry))) (object v11))) (agent v10))
  (p28 (class (m24 (lex person))) (member v10)))
 (cq (p31 (object v11) (property (m102 (lex small))))))

(m137!)

 CPU time : 0.03

*
;;; If a person wants something, then it is valuable
;;; (Ehrlich doesn't mention that *v9 must be a person, so I modify it)

(describe
(assert forall (*v13 *v9)
        &ant ((build agent *v9
                     act (build action (build lex "want") object *v13))
              (build member *v9
                     class (build lex "person")))
        cq (build object *v13
                  property (build lex "valuable"))
        ))

(m140! (forall v11 v10)
 (&ant (p33 (act (p32 (action (m138 (lex want))) (object v11))) (agent v10))
  (p28 (class (m24 (lex person))) (member v10)))
 (cq (p34 (object v11) (property (m139 (lex valuable))))))

(m140!)

 CPU time : 0.04

*
;;; If someone says they want something, then they do

(describe
(assert forall (*v13 *v9)
        ant (build act (build action (build lex "say that")
```

```
                              object (build act (build action (build lex "want") object *v13)
                                                agent *v9))
                     agent *v9)
        cq (build act (build action (build lex "want") object *v13)
                    agent *v9)
        ))

(m142! (forall v11 v10)
 (ant
  (p36
   (act (p35 (action (m141 (lex say that)))
         (object
          (p33 (act (p32 (action (m138 (lex want))) (object v11)))
           (agent v10)))))
   (agent v10)))
 (cq (p33)))

(m142!)

 CPU time : 0.02

*
;;; If something has color and belongs to some class, then that class is a
;;;    subclass of physical object

(describe
(assert forall ($v14 $v15 $v16)
        &ant ((build member *v15
                    class (build lex "color"))
              (build object *v16
                     property *v15)
              (build member *v16
                     class *v14))
        cq (build subclass *v14
                  superclass (build lex "phys obj"))
        ))

(m143! (forall v14 v13 v12)
 (&ant (p39 (class v12) (member v14)) (p38 (object v14) (property v13))
  (p37 (class (m98 (lex color))) (member v13)))
 (cq (p40 (subclass v12) (superclass (m38 (lex phys obj))))))

(m143!)

 CPU time : 0.02

*
;;; If something has size and belongs to some class, then that class is a
;;;    subclass of physical object

(describe
(assert forall (*v14 *v15 $v17)
        &ant ((build member *v15
                    class (build lex "size"))
              (build member *v17
                     class *v14)
              (build object *v17
                     property *v15))
```

```
        cq (build subclass *v14
                  superclass (build lex "phys obj"))
        ))

(m144! (forall v15 v13 v12)
 (&ant (p43 (object v15) (property v13)) (p42 (class v12) (member v15))
  (p41 (class (m103 (lex size))) (member v13)))
 (cq (p40 (subclass v12) (superclass (m38 (lex phys obj))))))

(m144!)

 CPU time : 0.02

*
;;; If a member of some class has a property, then it is possible for other
;;;    members of that class to have that property

(describe
(assert forall (*v14 *v15 *v17 $v18)
        &ant ((build object *v17
                     property *v15)
              (build member *v17
                     class *v14)
              (build member *v18
                     class *v14))
        cq (build mode (build lex "possibly")
                  object (build object *v18
                                property *v15))
        ))

(m146! (forall v16 v15 v13 v12)
 (&ant (p44 (class v12) (member v16)) (p43 (object v15) (property v13))
  (p42 (class v12) (member v15)))
 (cq
  (p46 (mode (m145 (lex possibly)))
   (object (p45 (object v16) (property v13))))))

(m146!)

 CPU time : 0.01

*
;;; If an animal acts, then the act performed is an action

(describe
(assert forall ($v19 $v20)
        &ant ((build member *v20
                     class (build lex "animal"))
              (build act (build action *v19)
                     agent *v20))
        cq (build member *v19
                  class (build lex "action"))
        ))

(m148! (forall v18 v17)
 (&ant (p49 (act (p48 (action v17))) (agent v18))
  (p47 (class (m18 (lex animal))) (member v18)))
 (cq (p50 (class (m147 (lex action))) (member v17))))
```

```
(m148!)

 CPU time : 0.02

*
;;; If a person acts, then the act performed is an action

(describe
(assert forall ($v21 $v22)
        &ant ((build member *v22
                     class (build lex "person"))
              (build act (build action *v21)
                     agent *v22))
        cq (build member *v21
                  class (build lex "action"))
        ))

(m149! (forall v20 v19)
 (&ant (p53 (act (p52 (action v19))) (agent v20))
  (p51 (class (m24 (lex person))) (member v20)))
 (cq (p54 (class (m147 (lex action))) (member v19))))

(m149!)

 CPU time : 0.02

*
;;; Spears are weapons

(describe
(assert subclass (build lex "spear")
        superclass (build lex "weapon")
        ))

(m152! (subclass (m150 (lex spear))) (superclass (m151 (lex weapon))))

(m152!)

 CPU time : 0.01

*
;;; If something is a weapon, then its function is to damage

(describe
(assert forall $v23
        ant (build member *v23
                   class (build lex "weapon")
                   )
        cq (build object1 *v23
                  object2 (build lex "damage")
                  rel (build lex "function"))
        ))

(m154! (forall v21) (ant (p55 (class (m151 (lex weapon))) (member v21)))
 (cq
  (p56 (object1 v21) (object2 (m153 (lex damage)))
   (rel (m118 (lex function))))))
```

```
(m154!)

 CPU time : 0.01

*
;;; Kings are persons

(describe
(assert subclass (build lex "king")
        superclass (build lex "person")
        ))

(m155! (subclass (m45 (lex king))) (superclass (m24 (lex person))))

(m155!)

 CPU time : 0.01

*
;;; Squires are persons

(describe
(assert subclass (build lex "squire")
        superclass (build lex "person")
        ))

(m157! (subclass (m156 (lex squire))) (superclass (m24 (lex person))))

(m157!)

 CPU time : 0.18

*
;;; Yeomen are persons

(describe
(assert subclass (build lex "yeoman")
        superclass (build lex "person")
        ))

(m159! (subclass (m158 (lex yeoman))) (superclass (m24 (lex person))))

(m159!)

 CPU time : 0.01

*
;;; If something is a carnivore, then it eats meat

(describe
(assert forall *v5
        ant (build member *v5
                  class (build lex "carnivore")
                  )
        cq (build act (build action (build lex "eat") object (build lex "meat"))
                  agent *v5)))
```

```
(m163! (forall v7) (ant (p57 (class (m14 (lex carnivore))) (member v7)))
 (cq
  (p58 (act (m162 (action (m160 (lex eat))) (object (m161 (lex meat)))))
   (agent v7))))

(m163!)

 CPU time : 0.02

*
;;; If something is an herbivore, then it eats plants and does not eat meat

(describe
(assert forall *v5
        ant (build member *v5
                   class (build lex "herbivore")
                   )
        cq ((build act (build action (build lex "eat") object (build lex "plant"))
                   agent *v5)
            (build min 0
                   max 0
                   arg (build act (build action (build lex "eat") object (build lex "meat"))
                              agent *v5)))))

(m166! (forall v7) (ant (p59 (class (m28 (lex herbivore))) (member v7)))
 (cq
  (p61 (min 0) (max 0)
   (arg
    (p58 (act (m162 (action (m160 (lex eat))) (object (m161 (lex meat)))))
     (agent v7))))
  (p60 (act (m165 (action (m160)) (object (m164 (lex plant))))) (agent v7))))

(m166!)

 CPU time : 0.01

*
;;; If something is a mammal, presumably it bears
;;; **NOTE** Change to bears live young
(describe
(assert forall *v5
        ant (build member *v5
                   class (build lex "mammal"))
        cq (build mode (build lex "presumably")
                  object (build act (build action (build lex "bear"))
                                agent *v5))))

(m169! (forall v7) (ant (p62 (class (m10 (lex mammal))) (member v7)))
 (cq
  (p64 (mode (m116 (lex presumably)))
   (object (p63 (act (m168 (action (m167 (lex bear)))) (agent v7)))))))

(m169!)

 CPU time : 0.03

*
;;;
```

```
;;; If something bears, then it is a mammal
;;; **NOTE** Change to bears live young
;;;(describe
;;;(assert forall ($v24  *v5)
;;;      ant (build act (build action (build lex "bear") object *v24)
;;;                  agent *v5)
;;;      cq (build member *v5
;;;                 class (build lex "mammal"))))

;;; If something bears and it is a member of a class then that class
;;;  is a subclass of mammal.
(describe
 (assert forall ($mammal $young $sup)
   &ant ((build agent *mammal
                act (build action (build lex "bear")
                           object *young))
         (build member *mammal class *sup))
   cq (build subclass *sup superclass (build lex "mammal"))))

(m170! (forall v24 v23 v22)
 (&ant (p67 (class v24) (member v22))
  (p66 (act (p65 (action (m167 (lex bear))) (object v23))) (agent v22)))
 (cq (p68 (subclass v24) (superclass (m10 (lex mammal))))))

(m170!)

 CPU time : 0.02

*
;;; If something is a predator, then presumably it hunts

(describe
(assert forall *v5
        ant (build member *v5
                  class (build lex "predator"))
        cq (build mode (build lex "presumably")
                  object (build act (build action (build lex "hunt"))
                                agent *v5))))

(m173! (forall v7) (ant (p69 (class (m16 (lex predator))) (member v7)))
 (cq
  (p71 (mode (m116 (lex presumably)))
   (object (p70 (act (m172 (action (m171 (lex hunt))))) (agent v7))))))

(m173!)

 CPU time : 0.02

*
;;; If an agent leaps onto an object at time x, then there is a time y when
;;;    the leaper is on the object, and y is after x

(describe
(assert forall ($v25 $v26 $v27)
        ant (build agent *v25
                  act (build action (build lex "leap")
                             onto *v26
                             time *v27))
```

```
        cq ((build object1 *v25
                   object2 *v26
                   rel (build lex "on")
                   time (build a3 *v26
                              skf "time-at"))
            (build after (build a3 *v26
                                skf "time-at")
                   before *v27))))

(m176! (forall v27 v26 v25)
 (ant
  (p73 (act (p72 (action (m174 (lex leap))) (onto v26) (time v27)))
   (agent v25)))
 (cq (p76 (after (p74 (a3 v26) (skf time-at))) (before v27))
  (p75 (object1 v25) (object2 v26) (rel (m175 (lex on))) (time (p74)))))

(m176!)

 CPU time : 0.04

*
;;; If an agent leaps to a goal at time x, then there is a time y when the
;;;    leaper is at the goal, and y is after x

(describe
(assert forall (*v25 *v26 *v27)
        ant (build agent *v25
                   act (build action (build lex "leap")
                              time *v27
                              to *v26))
        cq ((build object1 *v25
                   object2 *v26
                   rel (build lex "at")
                   time (build a3 *v26
                                skf "time-at"))
            (build after (build a3 *v26
                                skf "time-at")
                   before *v27))
        ))

(m178! (forall v27 v26 v25)
 (ant
  (p78 (act (p77 (action (m174 (lex leap))) (time v27) (to v26)))
   (agent v25)))
 (cq
  (p79 (object1 v25) (object2 v26) (rel (m177 (lex at)))
   (time (p74 (a3 v26) (skf time-at))))
  (p76 (after (p74)) (before v27))))

(m178!)

 CPU time : 0.03

*
;;; If an agent leaps from an object at time x, then there is a time y when
;;;    the leaper is on the object, and y is before x

(describe
```

```
(assert forall (*v25 *v26 $v28 $v29)
        ant (build agent *v25
                  act (build action (build lex "leap")
                            from *v28
                            time *v29))
        cq ((build object1 *v25
                  object2 *v28
                  rel (build lex "on")
                  time (build a4 *v28
                              skf "time-at"))
            (build after *v29
                  before (build a4 *v28
                                skf "time-at")))
        ))

(m179! (forall v29 v28 v26 v25)
 (ant
  (p81 (act (p80 (action (m174 (lex leap))) (from v28) (time v29)))
   (agent v25)))
 (cq (p84 (after v29) (before (p82 (a4 v28) (skf time-at))))
  (p83 (object1 v25) (object2 v28) (rel (m175 (lex on))) (time (p82)))))

(m179!)

 CPU time : 0.03

*

(describe
(assert forall ($v30 $v31 $v32)
        ant ((build agent *v30
                    place *v31
                    time *v32)
             (build object *v30
                    place *v31
                    time *v32))
        cq (build object1 *v30
                  object2 *v31
                  rel (build lex "at")
                  time *v32)
        ))

(m180! (forall v32 v31 v30)
 (ant (p86 (object v30) (place v31) (time v32))
  (p85 (agent v30) (place v31) (time v32)))
 (cq (p87 (object1 v30) (object2 v31) (rel (m177 (lex at))) (time v32))))

(m180!)

 CPU time : 0.03

*
;; Define a primitive action that does nothing
^(
--> define-primaction noop ()
     "Do nothing"
     (format t ""))
noop
```

```
 CPU time : 0.00

* ^(
-->attach-primaction noop noop)
t

 CPU time : 0.01

*
^(
-->attach-primaction snif snif believe believe)
t

 CPU time : 0.01

* (describe
 (assert
   forall ($x $y $z)
   &ant ((build member *x class *y)
         (build member *x class *z)
         (build object *y property (build lex "unknown")))
   cq (build
       when (build member *x class *y)
       do (build action snif
                 object1 ((build condition
                                 (build subclass *z superclass *y)
                                 then (build action noop))
                          (build else
                                 (build action believe
                                        object1
                                        (build mode (build lex "presumably")
                                               object
                                               (build subclass *y superclass *z)))))))))

(m183! (forall v35 v34 v33)
 (&ant (p90 (object v34) (property (m181 (lex unknown))))
  (p89 (class v35) (member v33)) (p88 (class v34) (member v33)))
 (cq
  (p98
   (do (p97
        (action snif)
        (object1
         (p96
          (else
           (p95 (action believe)
            (object1
             (p94 (mode (m116 (lex presumably)))
              (object (p93 (subclass v34) (superclass v35))))))))
         (p92 (condition (p91 (subclass v35) (superclass v34)))
          (then (m182 (action noop)))))))
   (when (p88)))))

(m183!)

 CPU time : 0.04

*
```

```
End of /projects/stn2/CVA/demos/cat.base demonstration.

 CPU time : 2.01

*
;;; Begin Reading Story

;;; There is a cat
(describe
(add member #Pye
     class (build lex "cat")))

(m246! (class (m147 (lex action))) (member (m160 (lex eat))))
(m245! (act (m244 (action (m160)))) (agent b1))
(m243! (act (m162 (action (m160)) (object (m161 (lex meat))))) (agent b1))
(m241! (class (m35 (lex vertebrate))) (member b1))
(m240! (class (m16 (lex predator))) (member b1))
(m239! (class (m14 (lex carnivore))) (member b1))
(m238! (class (m12 (lex quadruped))) (member b1))
(m237! (class (m10 (lex mammal))) (member b1))
(m236! (class (m78 (lex furniture))) (member (m40 (lex Round Table))))
(m235! (class (m38 (lex phys obj))) (member b1))
(m232! (class (m24 (lex person))) (member b11))
(m231! (class (m24)) (member b10))
(m230! (class (m24)) (member b9))
(m229! (class (m24)) (member b8))
(m228! (class (m24)) (member b7))
(m227! (class (m24)) (member b6))
(m226! (class (m24)) (member b5))
(m225! (class (m24)) (member b3))
(m224! (class (m18 (lex animal))) (member b1))
(m185! (class (m184 (lex cat))) (member b12))
(m159! (subclass (m158 (lex yeoman))) (superclass (m24)))
(m157! (subclass (m156 (lex squire))) (superclass (m24)))
(m155! (subclass (m45 (lex king))) (superclass (m24)))
(m115! (class (m112 (lex value))) (member (m114 (lex bad))))
(m113! (class (m112)) (member (m111 (lex good))))
(m108! (class (m103 (lex size))) (member (m107 (lex large))))
(m104! (class (m103)) (member (m102 (lex small))))
(m101! (class (m98 (lex color))) (member (m100 (lex black))))
(m99! (class (m98)) (member (m97 (lex white))))
(m96! (class (m90 (lex basic ctgy))) (member (m41 (lex table))))
(m95! (class (m90)) (member (m81 (lex chair))))
(m94! (class (m90)) (member (m2 (lex deer))))
(m93! (class (m90)) (member (m83 (lex horse))))
(m92! (class (m90)) (member (m8 (lex dog))))
(m91! (class (m90)) (member (m24)))
(m89! (subclass (m69 (lex knight))) (superclass (m24)))
(m76! (class (m69)) (member b11))
(m73! (class (m69)) (member b10))
(m70! (class (m69)) (member b9))
(m66! (class (m45)) (member b8))
(m63! (class (m45)) (member b7))
(m60! (class (m45)) (member b6))
(m57! (subclass (m55 (lex wizard))) (superclass (m24)))
(m56! (class (m55)) (member b5))
(m51! (class (m50 (lex sword))) (member b4))
```

```
(m46! (class (m45)) (member b3))
(m42! (class (m41)) (member (m40)))
(m25! (class (m24)) (member b2))
(m20! (class (m8)) (member b1))

(m246! m245! m243! m241! m240! m239! m238! m237! m236! m235! m232! m231! m230!
 m229! m228! m227! m226! m225! m224! m185! m159! m157! m155! m115! m113! m108!
 m104! m101! m99! m96! m95! m94! m93! m92! m91! m89! m76! m73! m70! m66! m63!
 m60! m57! m56! m51! m46! m42! m25! m20!)

 CPU time : 5.92

*
;;; Cat is an unknown word
(describe
(add object (build lex "cat")
     property (build lex "unknown")))

(m246! (class (m147 (lex action))) (member (m160 (lex eat))))
(m241! (class (m35 (lex vertebrate))) (member b1))
(m240! (class (m16 (lex predator))) (member b1))
(m239! (class (m14 (lex carnivore))) (member b1))
(m238! (class (m12 (lex quadruped))) (member b1))
(m237! (class (m10 (lex mammal))) (member b1))
(m236! (class (m78 (lex furniture))) (member (m40 (lex Round Table))))
(m235! (class (m38 (lex phys obj))) (member b1))
(m232! (class (m24 (lex person))) (member b11))
(m231! (class (m24)) (member b10))
(m230! (class (m24)) (member b9))
(m229! (class (m24)) (member b8))
(m228! (class (m24)) (member b7))
(m227! (class (m24)) (member b6))
(m226! (class (m24)) (member b5))
(m225! (class (m24)) (member b3))
(m224! (class (m18 (lex animal))) (member b1))
(m202! (object (m184 (lex cat))) (property (m181 (lex unknown))))
(m115! (class (m112 (lex value))) (member (m114 (lex bad))))
(m113! (class (m112)) (member (m111 (lex good))))
(m108! (class (m103 (lex size))) (member (m107 (lex large))))
(m104! (class (m103)) (member (m102 (lex small))))
(m101! (class (m98 (lex color))) (member (m100 (lex black))))
(m99! (class (m98)) (member (m97 (lex white))))
(m96! (class (m90 (lex basic ctgy))) (member (m41 (lex table))))
(m95! (class (m90)) (member (m81 (lex chair))))
(m94! (class (m90)) (member (m2 (lex deer))))
(m93! (class (m90)) (member (m83 (lex horse))))
(m92! (class (m90)) (member (m8 (lex dog))))
(m91! (class (m90)) (member (m24)))
(m76! (class (m69 (lex knight))) (member b11))
(m73! (class (m69)) (member b10))
(m70! (class (m69)) (member b9))
(m66! (class (m45 (lex king))) (member b8))
(m63! (class (m45)) (member b7))
(m60! (class (m45)) (member b6))
(m56! (class (m55 (lex wizard))) (member b5))
(m51! (class (m50 (lex sword))) (member b4))
(m46! (class (m45)) (member b3))
(m42! (class (m41)) (member (m40)))
```

```
(m25! (class (m24)) (member b2))
(m20! (class (m8)) (member b1))

(m246! m241! m240! m239! m238! m237! m236! m235! m232! m231! m230! m229! m228!
 m227! m226! m225! m224! m202! m115! m113! m108! m104! m101! m99! m96! m95!
 m94! m93! m92! m91! m76! m73! m70! m66! m63! m60! m56! m51! m46! m42! m25!
 m20!)

 CPU time : 2.23

*
;;; The cat is named Pyewacket
(describe
(assert object *Pye
        proper-name (build lex "Pyewacket")))

(m262! (object b12) (proper-name (m261 (lex Pyewacket))))

(m262!)

 CPU time : 0.02

*
;;; What is the meaning of cat?
^(
-->defineNoun "cat")
 Definition of cat:
 Named Individuals: Pyewacket,
nil

 CPU time : 4.18

*
;;; There is a person
(describe
(add member #Evelyn
     class (build lex "person")))

(m276! (class (m24 (lex person))) (member b13))
(m232! (class (m24)) (member b11))
(m231! (class (m24)) (member b10))
(m230! (class (m24)) (member b9))
(m229! (class (m24)) (member b8))
(m228! (class (m24)) (member b7))
(m227! (class (m24)) (member b6))
(m226! (class (m24)) (member b5))
(m225! (class (m24)) (member b3))
(m25! (class (m24)) (member b2))

(m276! m232! m231! m230! m229! m228! m227! m226! m225! m25!)

 CPU time : 6.56

*
;;; The person is named Evelyn
(describe
(assert object *Evelyn
        proper-name (build lex "Evelyn")))
```

```
(m337! (object b13) (proper-name (m336 (lex Evelyn))))

(m337!)

 CPU time : 0.00

*
;;; Pyewacket is Evelyn's cat
(describe
(add object *Pye
     possessor *Evelyn
     rel (build lex "cat")))

(m338! (object b12) (possessor b13) (rel (m184 (lex cat))))

(m338!)

 CPU time : 0.09

*
;;; What is the meaning of cat?
^(
-->defineNoun "cat")
 Definition of cat:
 Possessive: person,
 Named Individuals: Pyewacket,
nil

 CPU time : 0.56

*
;;; Pyewacket bears something
(describe
(add agent *Pye
     act (build action (build lex "bear")
                object #kittens1)))

(m340! (act (m339 (action (m167 (lex bear))) (object b14))) (agent b12))
(m274! (act (m168 (action (m167)))) (agent b12))
(m266! (subclass (m184 (lex cat))) (superclass (m10 (lex mammal))))

(m340! m274! m266!)

 CPU time : 0.13

*
;;; The thing that Pyewacket bears is a kitten
(describe
(add member *kittens1
     class (build lex "kitten")))

(m342! (class (m341 (lex kitten))) (member b14))

(m342!)

 CPU time : 5.56
```

```
*
;;; Kittens are cats
(describe
(add subclass (build lex "kitten")
     superclass (build lex "cat")))

(m385! (subclass (m341 (lex kitten))) (superclass (m184 (lex cat))))

(m385!)

 CPU time : 0.10

*
;;; Kittens are young
(describe
(add forall $kitty
        ant (build member *kitty  class (build lex "kitten"))
        cq  (build object *kitty property (build lex "young"))))

(m388! (object b14) (property (m386 (lex young))))
(m387! (forall v42) (ant (p318 (class (m341 (lex kitten))) (member v42)))
 (cq (p319 (object v42) (property (m386)))))
(m342! (class (m341)) (member b14))

(m388! m387! m342!)

 CPU time : 0.42

*
;;; What is the meaning of cat?
^(
-->defineNoun "cat")
 Definition of cat:
 Class Inclusions: mammal,
 Possible Actions: bear kitten,
 Possessive: person,
nil

 CPU time : 9.68

*
;;; Something is a cat
(describe
(assert member #Frisk
     class (build lex "cat")))

(m417! (class (m184 (lex cat))) (member b15))

(m417!)

 CPU time : 0.02

*
;;; That cat is named Frisky
(describe
(assert proper-name (build lex "Frisky")
        object *Frisk ))
```

```
(m419! (object b15) (proper-name (m418 (lex Frisky))))

(m419!)

 CPU time : 0.01

*
;;; What is the meaning of cat?
^(
-->defineNoun "cat")
 Definition of cat:
 Class Inclusions: mammal,
 Possible Actions: bear kitten,
 Possessive: person,
nil

 CPU time : 3.30

*
;;; Frisky sleeps in an something
(describe
(add agent *Frisk
     act (build action (build lex "sleep"))
     place #armch ))

(m442! (class (m147 (lex action))) (member (m127 (lex sleep))))
(m441! (class (m38 (lex phys obj))) (member b15))
(m440! (class (m18 (lex animal))) (member b15))
(m439! (act (m128 (action (m127)))) (agent b15) (place b16))
(m429! (act (m128)) (agent b15))

(m442! m441! m440! m439! m429!)

 CPU time : 1.19

*
;;; The thing that Frisky sleeps in is a chair
(describe
(assert member *armch
        class (build lex "chair")))

(m443! (class (m81 (lex chair))) (member b16))

(m443!)

 CPU time : 0.00

*
;;; The thing that Frisky sleeps in is an armchair
(describe
(assert member *armch
        class (build lex "armchair")))

(m445! (class (m444 (lex armchair))) (member b16))

(m445!)

 CPU time : 0.03
```

```
*
;;; Armchairs are chairs (THIS SHOULD BE BK -- SN)
(describe
(assert subclass (build lex "armchair")
        superclass (build lex "chair")))

(m446! (subclass (m444 (lex armchair))) (superclass (m81 (lex chair))))

(m446!)

 CPU time : 0.01

*
;;; What is the meaning of cat?
^(
-->defineNoun "cat")
 Definition of cat:
 Class Inclusions: mammal,
 Possible Actions: bear kitten, sleep,
 Possessive: person,
nil

 CPU time : 2.63

*
;;; If something is a cat, then presumably it purrs.
(describe
(add forall $cat
        ant (build member *cat  class (build lex "cat"))
        cq  (build mode (build lex "presumably")
                   object (build agent *cat act (build action (build lex "purr"))))))

(m456! (mode (m116 (lex presumably)))
 (object (m455 (act (m448 (action (m447 (lex purr))))) (agent b15))))
(m454! (mode (m116)) (object (m453 (act (m448)) (agent b14))))
(m452! (class (m184 (lex cat))) (member b14))
(m451! (mode (m116)) (object (m450 (act (m448)) (agent b12))))
(m449! (forall v64) (ant (p414 (class (m184)) (member v64)))
 (cq (p416 (mode (m116)) (object (p415 (act (m448)) (agent v64))))))
(m417! (class (m184)) (member b15))
(m185! (class (m184)) (member b12))

(m456! m454! m452! m451! m449! m417! m185!)

 CPU time : 1.21

*
;;; What is the meaning of cat?
^(
-->defineNoun "cat")
 Definition of cat:
 Class Inclusions: mammal,
 Probable Actions: purr,
 Possible Properties: young,
 Possessive: person,
nil
```

```
 CPU time : 3.33

*
;;; If something is a cat, then presumably it hunts
(describe
(add forall *cat
        ant (build object *cat  class (build lex "cat"))
        cq  (build mode (build lex "presumably")
                   object (build agent *cat act (build action (build lex "hunt"))))))

(m458! (forall v64) (ant (p433 (class (m184 (lex cat))) (object v64)))
 (cq
  (p435 (mode (m116 (lex presumably)))
   (object (p434 (act (m172 (action (m171 (lex hunt)))))) (agent v64))))))

(m458!)

 CPU time : 0.54

*
;;; What is the meaning of cat?
^(
-->defineNoun "cat")
 Definition of cat:
 Class Inclusions: mammal,
 Probable Actions: purr, hunt,
 Possible Properties: young,
 Possessive: person,
nil

 CPU time : 3.29

*
;;; Cats are mammals
(describe
(add subclass (build lex "cat")
     superclass (build lex "mammal")))

(m266! (subclass (m184 (lex cat))) (superclass (m10 (lex mammal))))

(m266!)

 CPU time : 0.02

*
;;; What is the meaning of cat?
^(
-->defineNoun "cat")
 Definition of cat:
 Class Inclusions: mammal,
 Probable Actions: purr, hunt,
 Possible Properties: young,
 Possessive: person,
nil

 CPU time : 3.60

*
```

```
;;; Cats are predators
(describe
(add subclass (build lex "cat")
     superclass (build lex "predator")))

(m460! (subclass (m341 (lex kitten))) (superclass (m16 (lex predator))))
(m459! (subclass (m184 (lex cat))) (superclass (m16)))

(m460! m459!)

 CPU time : 0.21

*
;;; What is the meaning of cat?
^(
-->defineNoun "cat")
 Definition of cat:
 Class Inclusions: predator, mammal,
 Probable Actions: purr,
 Possible Properties: young,
 Possessive: person,
nil

 CPU time : 9.22

*
;;; Cats are quadrupeds
(describe
(add subclass (build lex "cat")
     superclass (build lex "quadruped")))

(m473! (subclass (m341 (lex kitten))) (superclass (m12 (lex quadruped))))
(m472! (subclass (m184 (lex cat))) (superclass (m12)))

(m473! m472!)

 CPU time : 0.22

*
;;; What is the meaning of cat?
^(
-->defineNoun "cat")
 Definition of cat:
 Class Inclusions: predator, quadruped, mammal,
 Probable Actions: purr,
 Possible Properties: young,
 Possessive: person,
nil

 CPU time : 12.80

*
;;; If something is a cat, then presumably it has whiskers
(describe
(add forall *cat
        ant (build member *cat  class (build lex "cat"))
        cq ((build mode (build lex "presumably")
                   object (build part (build lex "whisker")
```

```
                                    whole *cat)))))

(m489! (mode (m116 (lex presumably)))
 (object (m488 (part (m482 (lex whisker))) (whole b12))))
(m487! (mode (m116)) (object (m486 (part (m482)) (whole b15))))
(m485! (mode (m116)) (object (m484 (part (m482)) (whole b14))))
(m483! (forall v64) (ant (p414 (class (m184 (lex cat))) (member v64)))
 (cq (p525 (mode (m116)) (object (p524 (part (m482)) (whole v64))))))

(m489! m487! m485! m483!)

 CPU time : 1.03

*
;;; What is the meaning of cat?
^(
-->defineNoun "cat")
 Definition of cat:
 Class Inclusions: predator, quadruped, mammal,
 Probable Structure: whisker,
 Probable Actions: purr,
 Possible Properties: young,
 Possessive: person,
nil

 CPU time : 9.98

*
;;; End Reading Story

End of /projects/stn2/CVA/demos/cat.demo demonstration.

 CPU time : 90.63

*
```

B.3 Hackney

```
Starting image '/util/acl62/composer'
  with no arguments
  in directory '/projects/stn2/CVA/demos/'
  on machine 'localhost'.

International Allegro CL Enterprise Edition
6.2 [Solaris] (Aug 15, 2002 14:24)
Copyright (C) 1985-2002, Franz Inc., Berkeley, CA, USA.  All Rights Reserved.

This development copy of Allegro CL is licensed to:
   [4549] SUNY/Buffalo, N. Campus

;; Optimization settings: safety 1, space 1, speed 1, debug 2.
;; For a complete description of all compiler switches given the current
;; optimization settings evaluate (explain-compiler-settings).
;;---
;; Current reader case mode: :case-sensitive-lower
cl-user(1): :ld lna
Error: "lna" does not exist, cannot load
  [condition type: file-does-not-exist-error]
```

```
Restart actions (select using :continue):
 0: retry the load of lna
 1: skip loading lna
 2: Abort entirely from this process.
[1] cl-user(2): :pop
cl-user(3): :cd ..
/projects/stn2/CVA/
cl-user(4): :ld lan
Error: "lan" does not exist, cannot load
  [condition type: file-does-not-exist-error]

Restart actions (select using :continue):
 0: retry the load of lan
 1: skip loading lan
 2: Abort entirely from this process.
[1] cl-user(5): :pop
cl-user(6): :ld lna
; Loading /projects/stn2/CVA/lna.cl
;   Loading /projects/snwiz/bin/sneps.lisp
Loading system SNePS...10% 20% 30% 40% 50% 60% 70% 80% 90% 100%
SNePS-2.6 [PL:0a 2002/09/30 22:37:46] loaded.
Type '(sneps)' or '(snepslog)' to get started.
;   Loading /projects/stn2/CVA/defun_noun.cl

   Welcome to SNePS-2.6 [PL:0a 2002/09/30 22:37:46]

Copyright (C) 1984--2002 by Research Foundation of
State University of New York. SNePS comes with ABSOLUTELY NO WARRANTY!
Type '(copyright)' for detailed copyright information.
Type '(demo)' for a list of example applications.

   4/3/2003 15:36:51

* (demo "demos/hackney.demo")

File /projects/stn2/CVA/demos/hackney.demo is now the source of input.

 CPU time : 0.03

* ;;; Reset the network
(resetnet t)

Net reset

 CPU time : 0.02

*
;;; Don't trace infer
^(
-->setq snip:*infertrace* nil)
nil

 CPU time : 0.00

*
;;; redefine function to return nil
;;; so that forward inference will not be limited
```

```
^(
-->defun broadcast-one-report (rep)
   (let (anysent)
     (do.chset (ch *OUTGOING-CHANNELS* anysent)
               (when (isopen.ch ch)
                 (setq anysent (or (try-to-send-report rep ch) anysent)))))
   nil)
broadcast-one-report

 CPU time : 0.00

*
;;; Load all valid relations
(demo "/projects/stn2/CVA/demos/rels")

File /projects/stn2/CVA/demos/rels is now the source of input.

 CPU time : 0.01

* (define a1 a2 a3 a4 after agent antonym associated before cause
    class direction equiv etime from in indobj instr into lex location
    manner member mode object on onto part place possessor proper-name
    property rel skf sp-rel stime subclass superclass synonym time to whole
    kn_cat)

(a1 a2 a3 a4 after agent antonym associated before cause class direction equiv
 etime from in indobj instr into lex location manner member mode object on
 onto part place possessor proper-name property rel skf sp-rel stime subclass
 superclass synonym time to whole kn_cat)

 CPU time : 0.15

*

End of /projects/stn2/CVA/demos/rels demonstration.

 CPU time : 0.16

*
;;; Compose paths
(demo "/projects/stn2/CVA/demos/paths")

File /projects/stn2/CVA/demos/paths is now the source of input.

 CPU time : 0.01

* ;;; Composition of certain paths

;;; Make Before Transitive
(define-path before (compose before (kstar (compose after- ! before))))
before implied by the path (compose before (kstar (compose after- ! before)))
before- implied by the path (compose (kstar (compose before- ! after)) before-)

 CPU time : 0.00

*
;;; Make After Transitive
(define-path after (compose after (kstar (compose before- ! after))))
```

```
after implied by the path (compose after (kstar (compose before- ! after)))
after- implied by the path (compose (kstar (compose after- ! before)) after-)

 CPU time : 0.01

*
;; If X is a member of the class Y and Y is a subclass of Z then
;;   X is a member of the class Z.
;; Example:  If Fido is a brachet and all brachets are hounds then
;;;  Fido is a hound.
(define-path class (compose class (kstar (compose subclass- ! superclass))))
class implied by the path (compose class
                           (kstar (compose subclass- ! superclass)))
class- implied by the path (compose (kstar (compose superclass- ! subclass))
                            class-)

 CPU time : 0.00

*
;; Make subclass transitive
(define-path subclass (compose subclass (kstar (compose superclass- ! subclass))))
subclass implied by the path (compose subclass
                              (kstar (compose superclass- ! subclass)))
subclass- implied by the path (compose
                               (kstar (compose subclass- ! superclass))
                               subclass-)

 CPU time : 0.00

*
;; Make member move over equiv arcs
(define-path member (compose member (kstar (compose equiv- ! equiv))))
member implied by the path (compose member (kstar (compose equiv- ! equiv)))
member- implied by the path (compose (kstar (compose equiv- ! equiv)) member-)

 CPU time : 0.00

*

End of /projects/stn2/CVA/demos/paths demonstration.

 CPU time : 0.02

*
;;; Load Background Knowledge
(demo "/projects/stn2/CVA/demos/hackney.base")

File /projects/stn2/CVA/demos/hackney.base is now the source of input.

 CPU time : 0.00

* ;;; Knowledge Base for Hackney demo -- Includes Knowledge Base present
;;; at the beginning of the Cat and Brachet demo

;;; Harts are deer
(describe
(assert subclass (build lex "hart")
        superclass (build lex "deer")
```

```
        kn_cat "life"))

(m3! (kn_cat life) (subclass (m1 (lex hart))) (superclass (m2 (lex deer))))

(m3!)

 CPU time : 0.01

*
;;; Halls are buildings
(describe
(assert subclass (build lex "hall")
        superclass (build lex "building")
        kn_cat "life"))

(m6! (kn_cat life) (subclass (m4 (lex hall))) (superclass (m5 (lex building))))

(m6!)

 CPU time : 0.02

*
;;; Hounds are dogs
(describe
(assert subclass (build lex "hound")
        superclass (build lex "dog")
        kn_cat "life"))

(m9! (kn_cat life) (subclass (m7 (lex hound))) (superclass (m8 (lex dog))))

(m9!)

 CPU time : 0.02

*
;;; Dogs are mammals
(describe
(assert subclass (build lex "dog")
        superclass (build lex "mammal")
        kn_cat "life"))

(m11! (kn_cat life) (subclass (m8 (lex dog))) (superclass (m10 (lex mammal))))

(m11!)

 CPU time : 0.00

*
;;; Dogs are quadrupeds
(describe
(assert subclass (build lex "dog")
        superclass (build lex "quadruped")
        kn_cat "life"))

(m13! (kn_cat life) (subclass (m8 (lex dog)))
 (superclass (m12 (lex quadruped))))

(m13!)
```

```
 CPU time : 0.00

*
;;; Dogs are carnivores
(describe
(assert subclass (build lex "dog")
        superclass (build lex "carnivore")
        kn_cat "life"))

(m15! (kn_cat life) (subclass (m8 (lex dog)))
 (superclass (m14 (lex carnivore))))

(m15!)

 CPU time : 0.01

*
;;; Dogs are predators
(describe
(assert subclass (build lex "dog")
        superclass (build lex "predator")
        kn_cat "life"))

(m17! (kn_cat life) (subclass (m8 (lex dog)))
 (superclass (m16 (lex predator))))

(m17!)

 CPU time : 0.01

*
;;; Dogs are animals
(describe
(assert subclass (build lex "dog")
        superclass (build lex "animal")
        kn_cat "life"))

(m19! (kn_cat life) (subclass (m8 (lex dog))) (superclass (m18 (lex animal))))

(m19!)

 CPU time : 0.01

*
;;; Rex is a dog
;;; (Ehrlich didn't have a kn_cat arc here, must have left off in error)
;;; (Ehrlich never had an object - proper-name case frame for Rex)

(describe
(assert member #Rex
        class (build lex "dog")
        kn_cat "life"))

(m20! (class (m8 (lex dog))) (kn_cat life) (member b1))

(m20!)
```

```
 CPU time : 0.00

*
(describe
(assert object *Rex
        proper-name (build lex "Rex")
        kn_cat "life"))

(m22! (kn_cat life) (object b1) (proper-name (m21 (lex Rex))))

(m22!)

 CPU time : 0.02

*
;;; Rex belongs to a person
;;; (Ehrlich never had a kn_cat arc for person, must have left off in error)

(describe
(assert object *Rex
        possessor #person1
        rel (build lex "dog")))

(m23! (object b1) (possessor b2) (rel (m8 (lex dog))))

(m23!)

 CPU time : 0.02

*
(describe
(assert member *person1
        class (build lex "person")
        kn_cat "life"))

(m25! (class (m24 (lex person))) (kn_cat life) (member b2))

(m25!)

 CPU time : 0.01

*
;;; Deer are mammals
(describe
(assert subclass (build lex "deer")
        superclass (build lex "mammal")
        kn_cat "life"))

(m26! (kn_cat life) (subclass (m2 (lex deer))) (superclass (m10 (lex mammal))))

(m26!)

 CPU time : 0.01

*
;;; Deer are quadrupeds
(describe
(assert subclass (build lex "deer")
```

```
        superclass (build lex "quadruped")
        kn_cat "life"))

(m27! (kn_cat life) (subclass (m2 (lex deer)))
 (superclass (m12 (lex quadruped))))

(m27!)

 CPU time : 0.00

*
;;; Deer are herbivores
(describe
(assert subclass (build lex "deer")
        superclass (build lex "herbivore")
        kn_cat "life"))

(m29! (kn_cat life) (subclass (m2 (lex deer)))
 (superclass (m28 (lex herbivore))))

(m29!)

 CPU time : 0.02

*
;;; Carnivores are animals
(describe
(assert subclass (build lex "carnivore")
        superclass (build lex "animal")
        kn_cat "life"))

(m30! (kn_cat life) (subclass (m14 (lex carnivore)))
 (superclass (m18 (lex animal))))

(m30!)

 CPU time : 0.01

*
;;; Predators are carnivores
(describe
(assert subclass (build lex "predator")
        superclass (build lex "carnivore")
        kn_cat "life"))

(m31! (kn_cat life) (subclass (m16 (lex predator)))
 (superclass (m14 (lex carnivore))))

(m31!)

 CPU time : 0.01

*
;;; "Herbivore" and "carnivore" are antonyms
(describe
(assert antonym (build lex "herbivore")
        antonym (build lex "carnivore")))
```

```
(m32! (antonym (m28 (lex herbivore)) (m14 (lex carnivore))))

(m32!)

 CPU time : 0.01

*
;;; "Predator" and "carnivore" are antonyms
(describe
(assert antonym (build lex "predator")
        antonym (build lex "carnivore")))

(m33! (antonym (m16 (lex predator)) (m14 (lex carnivore))))

(m33!)

 CPU time : 0.00

*
;;; Mammals are animals
(describe
(assert subclass (build lex "mammal")
        superclass (build lex "animal")
        kn_cat "life"))

(m34! (kn_cat life) (subclass (m10 (lex mammal)))
 (superclass (m18 (lex animal))))

(m34!)

 CPU time : 0.01

*
;;; Mammals are vertebrates
(describe
(assert subclass (build lex "mammal")
        superclass (build lex "animal")
        kn_cat "life"))

(m34! (kn_cat life) (subclass (m10 (lex mammal)))
 (superclass (m18 (lex animal))))

(m34!)

 CPU time : 0.02

*
;;; Quadrupeds are vertebrates
(describe
(assert subclass (build lex "quadruped")
        superclass (build lex "vertebrate")
        kn_cat "life"))

(m36! (kn_cat life) (subclass (m12 (lex quadruped)))
 (superclass (m35 (lex vertebrate))))

(m36!)
```

```
 CPU time : 0.03

*
;;; Animals are physical objects
(describe
(assert subclass (build lex "animal")
        superclass (build lex "phys obj")
        kn_cat "life"))

(m38! (kn_cat life) (subclass (m18 (lex animal)))
 (superclass (m37 (lex phys obj))))

(m38!)

 CPU time : 0.02

*
;;; The Round Table is a table
(describe
(assert member (build lex "Round Table")
        class (build lex "table")
        kn_cat "life"))

(m41! (class (m40 (lex table))) (kn_cat life) (member (m39 (lex Round Table))))

(m41!)

 CPU time : 0.01

*
;;; King Arthur is a king
(describe
(assert object #KA
        proper-name (build lex "King Arthur")
        kn_cat "life"))

(m43! (kn_cat life) (object b3) (proper-name (m42 (lex King Arthur))))

(m43!)

 CPU time : 0.01

*
(describe
(assert member *KA

        class (build lex "king")
        kn_cat "life"))

(m45! (class (m44 (lex king))) (kn_cat life) (member b3))

(m45!)

 CPU time : 0.00

*
;;; The Round Table is King Arthur's table
(describe
```

```
(assert object (build lex "Round Table")
        possessor *KA
        rel (build lex "table")
        kn_cat "life"))

(m46! (kn_cat life) (object (m39 (lex Round Table))) (possessor b3)
 (rel (m40 (lex table))))

(m46!)

 CPU time : 0.00

*
;;; Excalibur is a sword
(describe
(assert object #Excal
        proper-name (build lex "Excalibur")
        kn_cat "life"))

(m48! (kn_cat life) (object b4) (proper-name (m47 (lex Excalibur))))

(m48!)

 CPU time : 0.01

*
(describe
(assert member *Excal
        class (build lex "sword")
        kn_cat "life"))

(m50! (class (m49 (lex sword))) (kn_cat life) (member b4))

(m50!)

 CPU time : 0.01

*
;;; Excalibur is King Arthur's sword
(describe
(assert object *Excal
        possessor *KA
        rel (build lex "sword")
        kn_cat "life"))

(m51! (kn_cat life) (object b4) (possessor b3) (rel (m49 (lex sword))))

(m51!)

 CPU time : 0.00

*
;;; Merlin is a wizard
(describe
(assert object #Mer
        proper-name (build lex "Merlin")
        kn_cat "life"))
```

```
(m53! (kn_cat life) (object b5) (proper-name (m52 (lex Merlin))))

(m53!)

 CPU time : 0.01

*
(describe
(assert member *Mer

        class (build lex "wizard")
        kn_cat "life"))

(m55! (class (m54 (lex wizard))) (kn_cat life) (member b5))

(m55!)

 CPU time : 0.01

*
;;; Wizards are persons
(describe
(assert subclass (build lex "wizard")
        superclass (build lex "person")
        kn_cat "life"))

(m56! (kn_cat life) (subclass (m54 (lex wizard)))
 (superclass (m24 (lex person))))

(m56!)

 CPU time : 0.01

*
;;; King Ban is a king
(describe
(assert object #KingBan
        proper-name (build lex "King Ban")
        kn_cat "life"))

(m58! (kn_cat life) (object b6) (proper-name (m57 (lex King Ban))))

(m58!)

 CPU time : 0.01

*
(describe
(assert member *KingBan

        class (build lex "king")
        kn_cat "life"))

(m59! (class (m44 (lex king))) (kn_cat life) (member b6))

(m59!)

 CPU time : 0.02
```

```
*
;;; King Bors is a king
(describe
(assert object #KingBors
        proper-name (build lex "King Bors")
        kn_cat "life"))

(m61! (kn_cat life) (object b7) (proper-name (m60 (lex King Bors))))

(m61!)

 CPU time : 0.01

*
(describe
(assert member *KingBors

        class (build lex "king")
        kn_cat "life"))

(m62! (class (m44 (lex king))) (kn_cat life) (member b7))

(m62!)

 CPU time : 0.01

*
;;; King Lot is a king
(describe
(assert object #KingLot
        proper-name (build lex "King Lot")
        kn_cat "life"))

(m64! (kn_cat life) (object b8) (proper-name (m63 (lex King Lot))))

(m64!)

 CPU time : 0.02

*
(describe
(assert member *KingLot

        class (build lex "king")
        kn_cat "life"))

(m65! (class (m44 (lex king))) (kn_cat life) (member b8))

(m65!)

 CPU time : 0.01

*
;;; Sir Galahad is a knight
(describe
(assert object #SirGal
        proper-name (build lex "Sir Galahad")
```

```
        kn_cat "life"))

(m67! (kn_cat life) (object b9) (proper-name (m66 (lex Sir Galahad))))

(m67!)

 CPU time : 0.01

*
(describe
(assert member *SirGal

        class (build lex "knight")
        kn_cat "life"))

(m69! (class (m68 (lex knight))) (kn_cat life) (member b9))

(m69!)

 CPU time : 0.01

*
;;; Sir Gawain is a knight
(describe
(assert object #SG
        proper-name (build lex "Sir Gawain")
        kn_cat "life"))

(m71! (kn_cat life) (object b10) (proper-name (m70 (lex Sir Gawain))))

(m71!)

 CPU time : 0.02

*
(describe
(assert member *SG

        class (build lex "knight")
        kn_cat "life"))

(m72! (class (m68 (lex knight))) (kn_cat life) (member b10))

(m72!)

 CPU time : 0.01

*
;;; Sir Tristram is a knight
(describe
(assert object #Tris
        proper-name (build lex "Sir Tristram")
        kn_cat "life"))

(m74! (kn_cat life) (object b11) (proper-name (m73 (lex Sir Tristram))))

(m74!)
```

```
 CPU time : 0.00

*
(describe
(assert member *Tris

        class (build lex "knight")
        kn_cat "life"))

(m75! (class (m68 (lex knight))) (kn_cat life) (member b11))

(m75!)

 CPU time : 0.01

*
;;; Sideboards are furniture
(describe
(assert subclass (build lex "sideboard")
        superclass (build lex "furniture")
        kn_cat "life"))

(m78! (kn_cat life) (subclass (m76 (lex sideboard)))
 (superclass (m77 (lex furniture))))

(m78!)

 CPU time : 0.01

*
;;; Tables are furniture
(describe
(assert subclass (build lex "table")
        superclass (build lex "furniture")
        kn_cat "life"))

(m79! (kn_cat life) (subclass (m40 (lex table)))
 (superclass (m77 (lex furniture))))

(m79!)

 CPU time : 0.00

*
;;; Chairs are furniture
(describe
(assert subclass (build lex "chair")
        superclass (build lex "furniture")
        kn_cat "life"))

(m81! (kn_cat life) (subclass (m80 (lex chair)))
 (superclass (m77 (lex furniture))))

(m81!)

 CPU time : 0.01

*
```

```
;;; Horses are quadrupeds
(describe
(assert subclass (build lex "horse")
        superclass (build lex "quadruped")
        kn_cat "life"))

(m83! (kn_cat life) (subclass (m82 (lex horse)))
 (superclass (m12 (lex quadruped))))

(m83!)

 CPU time : 0.01

*
;;; Horses are herbivores
(describe
(assert subclass (build lex "horse")
        superclass (build lex "herbivore")
        kn_cat "life"))

(m84! (kn_cat life) (subclass (m82 (lex horse)))
 (superclass (m28 (lex herbivore))))

(m84!)

 CPU time : 0.01

*
;;; Ungulates are herbivores
(describe
(assert subclass (build lex "ungulate")
        superclass (build lex "herbivore")
        kn_cat "life"))

(m86! (kn_cat life) (subclass (m85 (lex ungulate)))
 (superclass (m28 (lex herbivore))))

(m86!)

 CPU time : 0.00

*
;;; Horses are animals
(describe
(assert subclass (build lex "horse")
        superclass (build lex "animal")
        kn_cat "life"))

(m87! (kn_cat life) (subclass (m82 (lex horse)))
 (superclass (m18 (lex animal))))

(m87!)

 CPU time : 0.01

*
;;; Knights are persons
(describe
```

```
(assert subclass (build lex "knight")
        superclass (build lex "person")
        kn_cat "life"))

(m88! (kn_cat life) (subclass (m68 (lex knight)))
 (superclass (m24 (lex person))))

(m88!)

 CPU time : 0.01

*
;;; "Person" is a basic-level category
(describe
(assert member (build lex "person")

        class (build lex "basic ctgy")
        kn_cat "life"))

(m90! (class (m89 (lex basic ctgy))) (kn_cat life) (member (m24 (lex person))))

(m90!)

 CPU time : 0.01

*
;;; "Dog" is a basic-level category
(describe
(assert member (build lex "dog")

        class (build lex "basic ctgy")
        kn_cat "life"))

(m91! (class (m89 (lex basic ctgy))) (kn_cat life) (member (m8 (lex dog))))

(m91!)

 CPU time : 0.01

*
;;; "Horse" is a basic-level category
(describe
(assert member (build lex "horse")

        class (build lex "basic ctgy")
        kn_cat "life"))

(m92! (class (m89 (lex basic ctgy))) (kn_cat life) (member (m82 (lex horse))))

(m92!)

 CPU time : 0.01

*
;;; "Deer" is a basic-level category
(describe
(assert member (build lex "deer")
```

```
        class (build lex "basic ctgy")
        kn_cat "life"))

(m93! (class (m89 (lex basic ctgy))) (kn_cat life) (member (m2 (lex deer))))

(m93!)

 CPU time : 0.01

*
;;; "Chair" is a basic-level category
(describe
(assert member (build lex "chair")

        class (build lex "basic ctgy")
        kn_cat "life"))

(m94! (class (m89 (lex basic ctgy))) (kn_cat life) (member (m80 (lex chair))))

(m94!)

 CPU time : 0.01

*
;;; "Table" is a basic-level category
(describe
(assert member (build lex "table")

        class (build lex "basic ctgy")
        kn_cat "life"))

(m95! (class (m89 (lex basic ctgy))) (kn_cat life) (member (m40 (lex table))))

(m95!)

 CPU time : 0.00

*
;;; White is a color
(describe
(assert member (build lex "white")
        class (build lex "color")
        kn_cat "life"))

(m98! (class (m97 (lex color))) (kn_cat life) (member (m96 (lex white))))

(m98!)

 CPU time : 0.01

*
;;; Black is a color
(describe
(assert member (build lex "black")
        class (build lex "color")
        kn_cat "life"))

(m100! (class (m97 (lex color))) (kn_cat life) (member (m99 (lex black))))
```

```
(m100!)

 CPU time : 0.01

*
;;; Small is a size
(describe
(assert member (build lex "small")
        class (build lex "size")
        kn_cat "life"))

(m103! (class (m102 (lex size))) (kn_cat life) (member (m101 (lex small))))

(m103!)

 CPU time : 0.02

*
;;; "Small" and "little" are synonyms
(describe
(assert synonym (build lex "small")
        synonym (build lex "little")
        kn_cat "life"))

(m105! (kn_cat life) (synonym (m104 (lex little)) (m101 (lex small))))

(m105!)

 CPU time : 0.01

*
;;; Large is a size
(describe
(assert member (build lex "large")
        class (build lex "size")
        kn_cat "life"))

(m107! (class (m102 (lex size))) (kn_cat life) (member (m106 (lex large))))

(m107!)

 CPU time : 0.03

*
;;; "Large" and "big" are synonyms
(describe
(assert synonym (build lex "large")
        synonym (build lex "big")
        kn_cat "life"))

(m109! (kn_cat life) (synonym (m108 (lex big)) (m106 (lex large))))

(m109!)

 CPU time : 0.01

*
```

```
;;; Good is a value
(describe
(assert member (build lex "good")
        class (build lex "value")
        kn_cat "life"))

(m112! (class (m111 (lex value))) (kn_cat life) (member (m110 (lex good))))

(m112!)

 CPU time : 0.04

*
;;; Bad is a value
(describe
(assert member (build lex "bad")
        class (build lex "value")
        kn_cat "life"))

(m114! (class (m111 (lex value))) (kn_cat life) (member (m113 (lex bad))))

(m114!)

 CPU time : 0.01

*
;;; If x is a horse, then presumably there is a person, y, such that the
;;;    function of x is to be ridden by y
(describe
(assert forall $horse1
        ant (build member *horse1 class (build lex "horse"))
        cq ((build mode (build lex "presumably")
                   object (build member (build skf "rider-of"
                                               a2 *horse1) = thisrider
                                class (build lex "person")))
            (build mode (build lex "presumably")
                   object (build agent *thisrider
                                act (build action (build lex "ride")
                                           object *horse1))))
        kn_cat "life-rule.1"))

(m117! (forall v1) (ant (p1 (class (m82 (lex horse))) (member v1)))
 (cq
  (p7 (mode (m115 (lex presumably)))
   (object
    (p6 (act (p5 (action (m116 (lex ride))) (object v1)))
     (agent (p2 (a2 v1) (skf rider-of))))))
  (p4 (mode (m115)) (object (p3 (class (m24 (lex person))) (member (p2))))))
 (kn_cat life-rule.1))

(m117!)

 CPU time : 0.04

*
;;; If a rider rides an animal that belongs to some class, then that class is
;;;    a subclass of equine
```

```
(describe
(assert forall ($v2 $v3)
        &ant ((build member *v2
                     class (build lex "animal"))
              (build member *v2
                     class *v3)
              (build member (build a2 *v2
                                  skf "rider-of")
                     class (build lex "person"))
              (build agent (build a2 *v2
                                  skf "rider-of")
                     act (build action (build lex "ride")
                                object *v2)))
        cq (build subclass *v3
                  superclass (build lex "equine"))
        kn_cat "life-rule.1"))

(m119! (forall v3 v2)
 (&ant
  (p13 (act (p12 (action (m116 (lex ride))) (object v2)))
   (agent (p10 (a2 v2) (skf rider-of))))
  (p11 (class (m24 (lex person))) (member (p10))) (p9 (class v3) (member v2))
  (p8 (class (m18 (lex animal))) (member v2)))
 (cq (p14 (subclass v3) (superclass (m118 (lex equine)))))
 (kn_cat life-rule.1))

(m119!)

 CPU time : 0.03

*

;;; If x is an animal and a member of some other named class y, and if
;;; there is a z that rides x, then y is a subclass of "equine"

(describe
(assert forall (*v2 $v39)
        &ant ((build member *v2
                     class (build lex "animal"))
              (build member *v2

                     class *v39)
              (build agent (build a2 *v2
                                  skf "rider-of")
                     act (build action (build lex "ride")
                                object *v2)))
        cq (build subclass *v39
                  superclass (build lex "equine"))
        kn_cat "life-rule.1"))

(m120! (forall v4 v2)
 (&ant (p15 (class v4) (member v2))
  (p13 (act (p12 (action (m116 (lex ride))) (object v2)))
   (agent (p10 (a2 v2) (skf rider-of))))
  (p8 (class (m18 (lex animal))) (member v2)))
 (cq (p16 (subclass v4) (superclass (m118 (lex equine)))))
 (kn_cat life-rule.1))
```

```
(m120!)

 CPU time : 0.02

*
;;; If something bites, then it's an animal

(describe
(assert forall $v5
        ant (build act (build action (build lex "bite"))
                   agent *v5)
        cq (build member *v5
                  class (build lex "animal"))
        kn_cat "life-rule.1"))

(m123! (forall v5)
 (ant (p17 (act (m122 (action (m121 (lex bite))))) (agent v5)))
 (cq (p18 (class (m18 (lex animal))) (member v5))) (kn_cat life-rule.1))

(m123!)

 CPU time : 0.01

*
;;; If something sleeps, then it's an animal

(describe
(assert forall *v5
        ant (build act (build action (build lex "sleep"))
                   agent *v5)
        cq (build member *v5
                  class (build lex "animal"))
        kn_cat "life-rule.1"))

(m126! (forall v5)
 (ant (p19 (act (m125 (action (m124 (lex sleep))))) (agent v5)))
 (cq (p18 (class (m18 (lex animal))) (member v5))) (kn_cat life-rule.1))

(m126!)

 CPU time : 0.02

*
;;; If something ambles, then it's an animal

(describe
(assert forall *v5
        ant (build property (build lex "ambling")
                   object *v5)
        cq (build member *v5
                  class (build lex "animal"))
        kn_cat "life-rule.1"))

(m128! (forall v5) (ant (p20 (object v5) (property (m127 (lex ambling)))))
 (cq (p18 (class (m18 (lex animal))) (member v5))) (kn_cat life-rule.1))

(m128!)
```

```
 CPU time : 0.01

*
;;; If something is an animal and a member of another class, then that class
;;;    is a subclass of animal

(describe
(assert forall (*v5 $v6)
        &ant ((build member *v5
                     class (build lex "animal"))
              (build member *v5
                     class *v6))
        cq (build subclass *v6
                  superclass (build lex "animal"))
        kn_cat "life-rule.1"))

(m129! (forall v6 v5)
 (&ant (p21 (class v6) (member v5))
  (p18 (class (m18 (lex animal))) (member v5)))
 (cq (p22 (subclass v6) (superclass (m18))) (kn_cat life-rule.1))

(m129!)

 CPU time : 0.00

*
;;; If something is presumed to be an animal and a member of another class,
;;;    then presumably that class is a subclass of animal

(describe
(assert forall (*v5 *v6)
        &ant ((build mode (build lex "presumably")
                     object (build member *v5
                                   class (build lex "animal")))
              (build member *v5
                     class *v6))
        cq (build mode (build lex "presumably")
                  object (build subclass *v6
                                superclass (build lex "animal")))
        kn_cat "life-rule.1"))

(m130! (forall v6 v5)
 (&ant
  (p23 (mode (m115 (lex presumably)))
   (object (p18 (class (m18 (lex animal))) (member v5))))
  (p21 (class v6) (member v5)))
 (cq (p24 (mode (m115)) (object (p22 (subclass v6) (superclass (m18))))))
 (kn_cat life-rule.1))

(m130!)

 CPU time : 0.02

*
;;; If something is a mammal and a member of another (non-superordinate) class,
;;;    then that class is a subclass of mammal

(describe
```

```
(assert forall (*v5 *v6)
        &ant ((build member *v5
                     class *v6)
              (build member *v5
                     class (build lex "mammal")
                     ))
        cq ((build subclass *v6
                   superclass (build lex "animal"))
            (build subclass *v6
                   superclass (build lex "mammal")))
        kn_cat "life-rule.1"))

(m131! (forall v6 v5)
 (&ant (p25 (class (m10 (lex mammal))) (member v5))
  (p21 (class v6) (member v5)))
 (cq (p26 (subclass v6) (superclass (m10)))
  (p22 (subclass v6) (superclass (m18 (lex animal)))))
 (kn_cat life-rule.1))

(m131!)

 CPU time : 0.02

*
;;; If a person can carry an object, then that object is small

(describe
(assert forall ($v13 *v9)
        &ant ((build member *v9
                     class (build lex "person"))
              (build agent *v9
                     act (build action (build lex "carry")
                                object *v13)))
        cq (build object *v13
                  property (build lex "small"))
        kn_cat "life-rule.2"))

(m134! (forall v7)
 (&ant (p28 (act (p27 (action (m133 (lex carry))) (object v7))))
  (m132 (class (m24 (lex person)))))
 (cq (p29 (object v7) (property (m101 (lex small))))) (kn_cat life-rule.2))

(m134!)

 CPU time : 0.02

*
;;; If a person wants something, then it is valuable
;;; (Ehrlich doesn't mention that *v9 must be a person, so I modify it)

(describe
(assert forall (*v13 *v9)
        ant (build agent *v9
                   act (build action (build lex "want")
                              object *v13))
        cq (build object *v13
                  property (build lex "valuable"))
        kn_cat "life-rule.2"))
```

```
(m137! (forall v7)
 (ant (p31 (act (p30 (action (m135 (lex want))) (object v7)))))
 (cq (p32 (object v7) (property (m136 (lex valuable))))) (kn_cat life-rule.2))

(m137!)

 CPU time : 0.01

*
;;; If someone says they want something, then they do
(describe
(assert forall ($thingy $person)
        ant (build agent *person act (build action (build lex "say that")
                                           object (build agent *person
                                                         act (build action (build lex "want")
                                                                   object *thingy))))
        cq (build agent *person
                  act (build action (build lex "want")
                             object *thingy))
        kn_cat "life-rule.2"))

(m139! (forall v9 v8)
 (ant
  (p36
   (act (p35 (action (m138 (lex say that)))
         (object
          (p34 (act (p33 (action (m135 (lex want))) (object v8)))
           (agent v9)))))
   (agent v9)))
 (cq (p34)) (kn_cat life-rule.2))

(m139!)

 CPU time : 0.03

*
;;; If something has color and belongs to some class, then that class is a
;;;    subclass of physical object

(describe
(assert forall ($v14 $v15 $v16)
        &ant ((build member *v15
                     class (build lex "color"))
              (build object *v16
                     property *v15)
              (build member *v16
                     class *v14))
        cq (build subclass *v14
                  superclass (build lex "phys obj"))
        kn_cat "intrinsic"))

(m140! (forall v12 v11 v10)
 (&ant (p39 (class v10) (member v12)) (p38 (object v12) (property v11))
  (p37 (class (m97 (lex color))) (member v11)))
 (cq (p40 (subclass v10) (superclass (m37 (lex phys obj)))))
 (kn_cat intrinsic))
```

```
(m140!)

 CPU time : 0.03

*
;;; If something has size and belongs to some class, then that class is a
;;;    subclass of physical object

(describe
(assert forall (*v14 *v15 $v17)
        &ant ((build member *v15
                     class (build lex "size"))
              (build member *v17
                     class *v14)
              (build object *v17
                     property *v15))
        cq (build subclass *v14
                  superclass (build lex "phys obj"))
        kn_cat "intrinsic"))

(m141! (forall v13 v11 v10)
 (&ant (p43 (object v13) (property v11)) (p42 (class v10) (member v13))
  (p41 (class (m102 (lex size))) (member v11)))
 (cq (p40 (subclass v10) (superclass (m37 (lex phys obj)))))
 (kn_cat intrinsic))

(m141!)

 CPU time : 0.02

*
;;; If a member of some class has a property, then it is possible for other
;;;    members of that class to have that property

(describe
(assert forall (*v14 *v15 *v17 $v18)
        &ant ((build object *v17
                     property *v15)
              (build member *v17
                     class *v14)
              (build member *v18
                     class *v14))
        cq (build mode (build lex "possibly")
                  object (build object *v18
                                property *v15))
        kn_cat "life-rule.1"))

(m143! (forall v14 v13 v11 v10)
 (&ant (p44 (class v10) (member v14)) (p43 (object v13) (property v11))
  (p42 (class v10) (member v13)))
 (cq
  (p46 (mode (m142 (lex possibly)))
   (object (p45 (object v14) (property v11)))))
 (kn_cat life-rule.1))

(m143!)

 CPU time : 0.03
```

```
*
;;; If an animal acts, then the act performed is an action

(describe
(assert forall ($v19 $v20)
        &ant ((build member *v20
                     class (build lex "animal"))
              (build act (build action *v19)
                     agent *v20))
        cq (build member *v19
                  class (build lex "action"))
        kn_cat "intrinsic"))

(m145! (forall v16 v15)
 (&ant (p49 (act (p48 (action v15))) (agent v16))
  (p47 (class (m18 (lex animal))) (member v16)))
 (cq (p50 (class (m144 (lex action))) (member v15))) (kn_cat intrinsic))

(m145!)

 CPU time : 0.03

*
;;; If a person acts, then the act performed is an action

(describe
(assert forall ($v21 $v22)
        &ant ((build member *v22
                     class (build lex "person"))
              (build act (build action *v21)
                     agent *v22))
        cq (build member *v21
                  class (build lex "action"))
        kn_cat "intrinsic"))

(m146! (forall v18 v17)
 (&ant (p53 (act (p52 (action v17))) (agent v18))
  (p51 (class (m24 (lex person))) (member v18)))
 (cq (p54 (class (m144 (lex action))) (member v17))) (kn_cat intrinsic))

(m146!)

 CPU time : 0.20

*
;;; Spears are weapons

(describe
(assert subclass (build lex "spear")
        superclass (build lex "weapon")
        kn_cat "life"))

(m149! (kn_cat life) (subclass (m147 (lex spear)))
 (superclass (m148 (lex weapon))))

(m149!)
```

```
 CPU time : 0.01

*
;;; If something is a weapon, then its function is to damage

(describe
(assert forall $v23
        ant (build member *v23
                  class (build lex "weapon")
                  )
        cq (build object1 *v23
                  object2 (build lex "damage")
                  rel (build lex "function"))
        kn_cat "life-rule.1"))

(m152! (forall v19) (ant (p55 (class (m148 (lex weapon))) (member v19)))
 (cq
  (p56 (object1 v19) (object2 (m150 (lex damage)))
   (rel (m151 (lex function)))))
 (kn_cat life-rule.1))

(m152!)

 CPU time : 0.02

*
;;; Kings are persons

(describe
(assert subclass (build lex "king")
        superclass (build lex "person")
        kn_cat "life"))

(m153! (kn_cat life) (subclass (m44 (lex king)))
 (superclass (m24 (lex person))))

(m153!)

 CPU time : 0.01

*
;;; Squires are persons

(describe
(assert subclass (build lex "squire")
        superclass (build lex "person")
        kn_cat "life"))

(m155! (kn_cat life) (subclass (m154 (lex squire)))
 (superclass (m24 (lex person))))

(m155!)

 CPU time : 0.01

*
;;; Yeomen are persons
```

```
(describe
(assert subclass (build lex "yeoman")
        superclass (build lex "person")
        kn_cat "life"))

(m157! (kn_cat life) (subclass (m156 (lex yeoman)))
 (superclass (m24 (lex person))))

(m157!)

 CPU time : 0.02

*
;;; If something is a carnivore, then it eats meat

(describe
(assert forall $carnivore
        ant (build member *carnivore
                  class (build lex "carnivore")
                  )
        cq (build agent *carnivore
                  act (build action (build lex "eat")
                            object (build lex "meat")))
        kn_cat "life-rule.1"))

(m161! (forall v20) (ant (p57 (class (m14 (lex carnivore))) (member v20)))
 (cq
  (p58 (act (m160 (action (m158 (lex eat))) (object (m159 (lex meat)))))
   (agent v20)))
 (kn_cat life-rule.1))

(m161!)

 CPU time : 0.01

*
;;; If something is an herbivore, then it eats plants

(describe
(assert forall $herb
        ant (build member *herb
                  class (build lex "herbivore")
                  )
        cq (build agent *herb
                  act (build action (build lex "eat")
                            object (build lex "plant")))
        kn_cat "life-rule.1"))

(m164! (forall v21) (ant (p59 (class (m28 (lex herbivore))) (member v21)))
 (cq
  (p60 (act (m163 (action (m158 (lex eat))) (object (m162 (lex plant)))))
   (agent v21)))
 (kn_cat life-rule.1))

(m164!)

 CPU time : 0.01
```

```
*
;"mammals presumably bear live young"
(describe
(assert forall $animal1
        ant (build member *animal1  class (build lex "mammal"))
        cq (build mode (build lex "presumably")
                  object (build agent *animal1 act (build action (build lex "bear")
                                                          object (build lex "live young"))))
        kn_cat "life-rule.1"))

(m168! (forall v22) (ant (p61 (class (m10 (lex mammal))) (member v22)))
 (cq
  (p63 (mode (m115 (lex presumably)))
   (object
    (p62
     (act (m167 (action (m165 (lex bear))) (object (m166 (lex live young)))))
     (agent v22)))))
 (kn_cat life-rule.1))

(m168!)

 CPU time : 0.04

*
;"if something bears something else, it is an animal"
(describe
(assert forall (*animal1 $animal2)
        ant (build agent *animal1 act (build action (build lex "bear")
                                            object *animal2))
        cq (build member *animal1  class (build lex "mammal"))
        kn_cat "life-rule.1"))

(m169! (forall v23 v22)
 (ant (p65 (act (p64 (action (m165 (lex bear))) (object v23))) (agent v22)))
 (cq (p61 (class (m10 (lex mammal))) (member v22))) (kn_cat life-rule.1))

(m169!)

 CPU time : 0.02

*
;;; If something is a predator, then presumably it hunts

(describe
(assert forall *v5
        ant (build member *v5
                  class (build lex "predator")
                  )
        cq (build mode (build lex "presumably")
                  object (build act (build action (build lex "hunt"))
                                agent *v5))
        kn_cat "life-rule.1"))

(m172! (forall v5) (ant (p66 (class (m16 (lex predator))) (member v5)))
 (cq
  (p68 (mode (m115 (lex presumably)))
   (object (p67 (act (m171 (action (m170 (lex hunt)))))) (agent v5)))))
 (kn_cat life-rule.1))
```

```
(m172!)

 CPU time : 0.02

*

(describe
(assert forall ($v30 $v31 $v32)
        ant ((build agent *v30
                    place *v31
                    time *v32)
             (build object *v30
                    place *v31
                    time *v32))
        cq (build object1 *v30
                  object2 *v31
                  rel (build lex "at")
                  time *v32)
        kn_cat "life-rule.1"))

(m174! (forall v26 v25 v24)
 (ant (p70 (object v24) (place v25) (time v26))
  (p69 (agent v24) (place v25) (time v26)))
 (cq (p71 (object1 v24) (object2 v25) (rel (m173 (lex at))) (time v26)))
 (kn_cat life-rule.1))

(m174!)

 CPU time : 0.01

*

(describe
(assert forall ($v33 $v34 *v5 *v6)
        &ant ((build member *v5
                    class *v6)
              (build agent *v5
                     act (build action *v34))
              (build forall *v5
                     ant (build agent *v5
                                act (build action *v34))
                     cq (build member *v5
                               class *v33)))
        cq (build mode (build lex "presumably")
                  object (build subclass *v6
                                 superclass *v33))
        kn_cat "life-rule.2"))

(m175! (forall v28 v27 v6 v5)
 (&ant
  (p75 (forall v5) (ant (p73 (act (p72 (action v28))) (agent v5)))
   (cq (p74 (class v27) (member v5))))
  (p73) (p21 (class v6) (member v5)))
 (cq
  (p77 (mode (m115 (lex presumably)))
   (object (p76 (subclass v6) (superclass v27)))))
 (kn_cat life-rule.2))
```

```
(m175!)

 CPU time : 0.03

*
;;; "Kill" and "Slay" are synonyms

(describe
(assert synonym "kill"
        synonym "slay"
        kn_cat "life"))

(m176! (kn_cat life) (synonym slay kill))

(m176!)

 CPU time : 0.01

*
;;; If x is an elder, then x has the property "old," and presumably x is a
;;; person

(describe
(assert forall $v35
        ant (build member *v35
                   class (build lex "elder")
                   )
        cq ((build object *v35
                   property (build lex "old"))
            (build mode (build lex "presumably")
                   object (build member *v35
                                 class (build lex "person"))))
        kn_cat "life-rule.1"))

(m179! (forall v29) (ant (p78 (class (m177 (lex elder))) (member v29)))
 (cq
  (p81 (mode (m115 (lex presumably)))
   (object (p80 (class (m24 (lex person))) (member v29))))
  (p79 (object v29) (property (m178 (lex old)))))
 (kn_cat life-rule.1))

(m179!)

 CPU time : 0.02

*
;;; Dwarfs are persons

(describe
(assert subclass (build lex "dwarf")
        superclass (build lex "person")
        kn_cat "life"))

(m181! (kn_cat life) (subclass (m180 (lex dwarf)))
 (superclass (m24 (lex person))))

(m181!)
```

```
 CPU time : 0.01

*
;;; Pavilions are tents

(describe
(assert subclass (build lex "pavilion")
        superclass (build lex "tent")
        kn_cat "life"))

(m184! (kn_cat life) (subclass (m182 (lex pavilion)))
 (superclass (m183 (lex tent))))

(m184!)

 CPU time : 0.01

*
;;; For all x and y, if there is a z that is the "rider-of" x and z is
;;; equivalent to y, then y rides x
(describe
(assert forall (*v2 $v38)
        ant (build equiv (build a2 *v2
                                skf "rider-of")
                  equiv  *v38)
        cq (build agent *v38
                  act (build action (build lex "ride")
                            object *v2))
        kn_cat "life-rule.1"))

(m185! (forall v30 v2) (ant (p82 (equiv v30 (p10 (a2 v2) (skf rider-of)))))
 (cq (p83 (act (p12 (action (m116 (lex ride))) (object v2))) (agent v30)))
 (kn_cat life-rule.1))

(m185!)

 CPU time : 0.01

*
;;; If y rides x, then there is a z that is the "rider-of" x, and z, is
;;; equivalent to y and z rides x

(describe
(assert forall (*v2 *v38)
        ant (build agent *v38
                  act (build action (build lex "ride")
                            object *v2))
        cq ((build equiv (build a2 *v2
                                skf "rider-of")
                  equiv *v38)
            (build agent (build a2 *v2
                                skf "rider-of")
                  act (build action (build lex "ride")
                            object *v2)))
        kn_cat "life-rule.1"))

(m186! (forall v30 v2)
```

```
 (ant (p83 (act (p12 (action (m116 (lex ride))) (object v2))) (agent v30)))
 (cq (p82 (equiv v30 (p10 (a2 v2) (skf rider-of))))
  (p13 (act (p12)) (agent (p10))))
 (kn_cat life-rule.1))

(m186!)

 CPU time : 0.03

*
;;; If y mounts x, then presumably y rides x

(describe
(assert forall (*v2 *v38)
        ant (build agent *v38
                   act (build action (build lex "mount")
                              object *v2))
        cq (build mode (build lex "presumably")
                  object (build agent *v38
                                act (build action (build lex "ride")
                                           object *v2)))
        kn_cat "life-rule.1"))

(m188! (forall v30 v2)
 (ant (p85 (act (p84 (action (m187 (lex mount))) (object v2))) (agent v30)))
 (cq
  (p86 (mode (m115 (lex presumably)))
   (object
    (p83 (act (p12 (action (m116 (lex ride))) (object v2))) (agent v30)))))
 (kn_cat life-rule.1))

(m188!)

 CPU time : 0.03

*
;;; Horses are equines

(describe
(assert subclass (build lex "horse")
        superclass (build lex "equine")
        kn_cat "life"))

(m189! (kn_cat life) (subclass (m82 (lex horse)))
 (superclass (m118 (lex equine))))

(m189!)

 CPU time : 0.01

*
;;; Equines are herbivores

(describe
(assert subclass (build lex "equine")
        superclass (build lex "herbivore")
        kn_cat "life"))
```

```
(m190! (kn_cat life) (subclass (m118 (lex equine)))
 (superclass (m28 (lex herbivore))))

(m190!)

 CPU time : 0.00

*
;;; Equines are mammals

(describe
(assert subclass (build lex "equine")
        superclass (build lex "mammal")
        kn_cat "life"))

(m191! (kn_cat life) (subclass (m118 (lex equine)))
 (superclass (m10 (lex mammal))))

(m191!)

 CPU time : 0.01

*
;;; Equines are quadrupeds

(describe
(assert subclass (build lex "equine")
        superclass (build lex "quadruped")
        kn_cat "life"))

(m192! (kn_cat life) (subclass (m118 (lex equine)))
 (superclass (m12 (lex quadruped))))

(m192!)

 CPU time : 0.01

*

;;; Ponies are equines

(describe
(assert subclass (build lex "pony")
        superclass (build lex "equine")
        kn_cat "life"))

(m194! (kn_cat life) (subclass (m193 (lex pony)))
 (superclass (m118 (lex equine))))

(m194!)

 CPU time : 0.02

*
;;; Ponies are animals
(describe
 (assert subclass (build lex "pony")
         superclass (build lex "animal")
```

```
         kn_cat "life"))

(m195! (kn_cat life) (subclass (m193 (lex pony)))
 (superclass (m18 (lex animal))))

(m195!)

 CPU time : 0.01

*
;;; Misty is a pony

(describe
(assert member #Misty
        class (build lex "pony")
        kn_cat "life"))

(m196! (class (m193 (lex pony))) (kn_cat life) (member b12))

(m196!)

 CPU time : 0.01

*
(describe
(assert object *Misty
        proper-name (build lex "Misty")
        kn_cat "life"))

(m198! (kn_cat life) (object b12) (proper-name (m197 (lex Misty))))

(m198!)

 CPU time : 0.02

*
;;; Misty is owned by a person

(describe
(assert object *Misty
        possessor #APers
        rel (build lex "pony")))

(m199! (object b12) (possessor b13) (rel (m193 (lex pony))))

(m199!)

 CPU time : 0.01

*
(describe
(assert member #APers
        class (build lex "person")
        kn_cat "life"))

(m200! (class (m24 (lex person))) (kn_cat life) (member b14))

(m200!)
```

```
 CPU time : 0.02

*
;;; Mr. Ed is a horse

(describe
(assert member #MREd

        class (build lex "horse")
        kn_cat "life"))

(m201! (class (m82 (lex horse))) (kn_cat life) (member b15))

(m201!)

 CPU time : 0.01

*
(describe
(assert object *MREd
        proper-name (build lex "Mr Ed")
        kn_cat "life"))

(m203! (kn_cat life) (object b15) (proper-name (m202 (lex Mr Ed))))

(m203!)

 CPU time : 0.02

*
;;; Mr. Ed is owned by a person

(describe
(assert object *MREd
        possessor #APers2
        rel (build lex "horse")
        kn_cat "life"))

(m204! (kn_cat life) (object b15) (possessor b16) (rel (m82 (lex horse))))

(m204!)

 CPU time : 0.01

*
(describe
(assert member *APers2
        class (build lex "person")
        kn_cat "life"))

(m205! (class (m24 (lex person))) (kn_cat life) (member b16))

(m205!)

 CPU time : 0.01

*
```

```
;;; If x is the y of its possessor, then x is a y

(describe
(assert forall (*v39 $v40)
        ant (build object *v40
                   rel *v39)
        cq (build member *v40
                  class *v39)
        kn_cat "intrinsic"))

(m206! (forall v31 v4) (ant (p87 (object v31) (rel v4)))
 (cq (p88 (class v4) (member v31))) (kn_cat intrinsic))

(m206!)

 CPU time : 0.01

*
;;; If x is an equine, then there is a y that is the breast
;;; of x

(describe
(assert forall *v4
        ant (build member *v4
                   class (build lex "equine"))
        cq (build object (build a1 *v4
                                skf "breast-of")
                  possessor *v4
                  rel (build lex "breast"))
        kn_cat "life-rule.1"))

(m211! (ant (m207 (class (m118 (lex equine)))))
 (cq (m210 (object (m208 (skf breast-of))) (rel (m209 (lex breast)))))
 (kn_cat life-rule.1))

(m211!)

 CPU time : 0.02

*
;;; RULES CONCERNING LEAPING

;;; If an agent leaps onto an object at time x, then there is a time y when
;;;    the leaper is on the object, and y is after x

(describe
(assert forall ($v25 $v26 $v27)
        ant (build agent *v25
                   act (build action (build lex "leap onto")
                             object *v26)
                   time *v27)
        cq ((build object *v25 location
                   (build sp-rel (build lex "on") object *v26)
                   time (build a3 *v26
                               skf "time-at"))
            (build after (build a3 *v26
                                skf "time-at")
                   before *v27))
```

```
        kn_cat "life-rule.1"))

(m214! (forall v34 v33 v32)
 (ant
  (p90 (act (p89 (action (m212 (lex leap onto))) (object v33))) (agent v32)
   (time v34)))
 (cq (p94 (after (p92 (a3 v33) (skf time-at))) (before v34))
  (p93 (location (p91 (object v33) (sp-rel (m213 (lex on))))) (object v32)
   (time (p92))))
 (kn_cat life-rule.1))

(m214!)

 CPU time : 0.03

*
;;; If an agent leaps to a goal at time x, then there is a time y when the
;;;    leaper is at the goal, and y is after x

(describe
(assert forall (*v25 *v26 *v27)
        ant (build agent *v25
                   act (build action (build lex "leap to")
                             object *v26)
                   time *v27)
        cq ((build object *v25 location
                   (build sp-rel (build lex "at") object *v26)
                   time (build a3 *v26
                               skf "time-at"))
            (build after (build a3 *v26
                                skf "time-at")
                   before *v27))
        kn_cat "life-rule.1"))

(m216! (forall v34 v33 v32)
 (ant
  (p96 (act (p95 (action (m215 (lex leap to))) (object v33))) (agent v32)
   (time v34)))
 (cq
  (p98 (location (p97 (object v33) (sp-rel (m173 (lex at))))) (object v32)
   (time (p92 (a3 v33) (skf time-at))))
  (p94 (after (p92)) (before v34)))
 (kn_cat life-rule.1))

(m216!)

 CPU time : 0.03

*
;;; If an agent leaps from an object at time x, then there is a time y when
;;;    the leaper is on the object, and y is before x

(describe
(assert forall (*v25 *v26 $v28 $v29)
        ant (build agent *v25
                   act (build action (build lex "leap from")
                             object *v28)
                   time *v29)
```

```
        cq ((build object *v25 location
                   (build sp-rel (build lex "on") object *v28)
                   time (build a4 *v28
                                skf "time-at"))
            (build after *v29
                   before (build a4 *v28
                                skf "time-at")))
        kn_cat "life-rule.1"))

(m218! (forall v36 v35 v33 v32)
 (ant
  (p100 (act (p99 (action (m217 (lex leap from))) (object v35))) (agent v32)
   (time v36)))
 (cq (p104 (after v36) (before (p102 (a4 v35) (skf time-at))))
  (p103 (location (p101 (object v35) (sp-rel (m213 (lex on))))) (object v32)
   (time (p102))))
 (kn_cat life-rule.1))

(m218!)

 CPU time : 0.01

*
;;;; NEW RULES -- SN

;;; If an agent leaps onto an object at time S and
;;;   the same agent leaps off of another object at time T and
;;;   time S is before time T then
;;;   the two objects are equivalent
;;;(describe
;;;(add forall ($leaper $thing1 $thing2 $timeS $timeT $cls1 $cls2)
;;;     &ant ((build agent *leaper
;;;                  act (build action (build lex "leap onto")
;;;                            object *thing1)
;;;                  time *timeS)
;;;        (build agent *leaper
;;;                  act (build action (build lex "leap from")
;;;                            object *thing2)
;;;                  time *timeT)
;;;           (build before *timeS after *timeT)
;;;           (build member *thing1 class *cls1)
;;;           (build member *thing2 class *cls2)
;;;           (build object *cls1 property (build lex "unknown")))
;;;     cq ((build equiv *thing1 equiv *thing2)
;;;         (build subclass *cls1 superclass *cls2))))

(describe
 (add forall ($leaper $thing1 $thing2 $timeS $timeT)
      &ant ((build agent *leaper
                  act (build action (build lex "leap onto")
                            object *thing1)
                  time *timeS)
            (build agent *leaper
                  act (build action (build lex "leap from")
                            object *thing2)
                  time *timeT)
            (build before *timeS after *timeT))
      cq (build equiv *thing1 equiv *thing2)))
```

```
(m219! (forall v41 v40 v39 v38 v37)
 (&ant (p109 (after v41) (before v40))
  (p108 (act (p107 (action (m217 (lex leap from))) (object v39))) (agent v37)
   (time v41))
  (p106 (act (p105 (action (m212 (lex leap onto))) (object v38))) (agent v37)
   (time v40)))
 (cq (p110 (equiv v39 v38))))

(m219!)

 CPU time : 0.45

*
(describe
(add forall ($eq1 $eq2 $cls1 $cls2)
     &ant ((build equiv *eq1 equiv *eq2)
           (build member *eq1 class *cls1)
           (build member *eq2 class *cls2)
           (build object *cls1 property (build lex "unknown")))
     cq (build subclass *cls1 superclass *cls2)))

(m293! (class (m209 (lex breast))) (member (m208 (skf breast-of))))
(m292! (act (m286 (action (m158 (lex eat))))) (agent b12))
(m291! (act (m286)) (agent b15))
(m290! (act (m163 (action (m158)) (object (m162 (lex plant))))) (agent b12))
(m289! (act (m163)) (agent b15))
(m288! (class (m144 (lex action))) (member (m158)))
(m287! (act (m286)) (agent b1))
(m285! (act (m160 (action (m158)) (object (m159 (lex meat))))) (agent b1))
(m284! (object b15) (rel (m82 (lex horse))))
(m283! (object b12) (rel (m193 (lex pony))))
(m282! (object b4) (rel (m49 (lex sword))))
(m281! (object (m39 (lex Round Table))) (rel (m40 (lex table))))
(m280! (object b1) (rel (m8 (lex dog))))
(m279! (class (m37 (lex phys obj))) (member b1))
(m278! (class (m35 (lex vertebrate))) (member b1))
(m277! (class (m18 (lex animal))) (member b1))
(m276! (class (m16 (lex predator))) (member b1))
(m275! (class (m14 (lex carnivore))) (member b1))
(m274! (class (m12 (lex quadruped))) (member b1))
(m273! (class (m10 (lex mammal))) (member b1))
(m272! (class (m8)) (member b1))
(m271! (class (m24 (lex person))) (member b2))
(m270! (class (m77 (lex furniture))) (member (m39)))
(m269! (class (m40)) (member (m39)))
(m268! (class (m44 (lex king))) (member b3))
(m267! (class (m24)) (member b3))
(m266! (class (m49)) (member b4))
(m265! (class (m54 (lex wizard))) (member b5))
(m264! (class (m24)) (member b5))
(m263! (class (m44)) (member b6))
(m262! (class (m24)) (member b6))
(m261! (class (m44)) (member b7))
(m260! (class (m24)) (member b7))
(m259! (class (m44)) (member b8))
(m258! (class (m24)) (member b8))
(m257! (class (m68 (lex knight))) (member b9))
```

```
(m256! (class (m24)) (member b9))
(m255! (class (m68)) (member b10))
(m254! (class (m24)) (member b10))
(m253! (class (m68)) (member b11))
(m252! (class (m24)) (member b11))
(m251! (class (m89 (lex basic ctgy))) (member (m24)))
(m250! (class (m89)) (member (m8)))
(m249! (class (m89)) (member (m82)))
(m248! (class (m89)) (member (m2 (lex deer))))
(m247! (class (m89)) (member (m80 (lex chair))))
(m246! (class (m89)) (member (m40)))
(m245! (class (m97 (lex color))) (member (m96 (lex white))))
(m244! (class (m97)) (member (m99 (lex black))))
(m243! (class (m102 (lex size))) (member (m101 (lex small))))
(m242! (class (m102)) (member (m106 (lex large))))
(m241! (class (m111 (lex value))) (member (m110 (lex good))))
(m240! (class (m111)) (member (m113 (lex bad))))
(m239! (class (m193)) (member b12))
(m238! (class (m118 (lex equine))) (member b12))
(m237! (class (m37)) (member b12))
(m236! (class (m35)) (member b12))
(m235! (class (m28 (lex herbivore))) (member b12))
(m234! (class (m18)) (member b12))
(m233! (class (m12)) (member b12))
(m232! (class (m10)) (member b12))
(m231! (class (m24)) (member b14))
(m230! (class (m118)) (member b15))
(m229! (class (m82)) (member b15))
(m228! (class (m37)) (member b15))
(m227! (class (m35)) (member b15))
(m226! (class (m28)) (member b15))
(m225! (class (m18)) (member b15))
(m224! (class (m12)) (member b15))
(m223! (class (m10)) (member b15))
(m222! (class (m24)) (member b16))
(m221! (forall v45 v44 v43 v42)
 (&ant (p114 (object v44) (property (m220 (lex unknown))))
  (p113 (class v45) (member v43)) (p112 (class v44) (member v42))
  (p111 (equiv v43 v42)))
 (cq (p115 (subclass v44) (superclass v45))))
(m210! (object (m208)) (rel (m209)))
(m207! (class (m118)))

(m293! m292! m291! m290! m289! m288! m287! m285! m284! m283! m282! m281! m280!
 m279! m278! m277! m276! m275! m274! m273! m272! m271! m270! m269! m268! m267!
 m266! m265! m264! m263! m262! m261! m260! m259! m258! m257! m256! m255! m254!
 m253! m252! m251! m250! m249! m248! m247! m246! m245! m244! m243! m242! m241!
 m240! m239! m238! m237! m236! m235! m234! m233! m232! m231! m230! m229! m228!
 m227! m226! m225! m224! m223! m222! m221! m210! m207!)

 CPU time : 5.41

*

End of /projects/stn2/CVA/demos/hackney.base demonstration.

 CPU time : 8.25
```

```
*
;;; Begin Reading Story

;;; When King Arthur and the two kings saw them begin to wax wroth on
;;; both sides, they leaped on small hackneys [...] [p 16]

(clear-infer)

(Node activation cleared. Some register information retained.)

 CPU time : 0.06

*
;;; King Arthur leaped onto something
(describe
(add agent *KA
        act (build action (build lex "leap onto")
                  object #hack1)
        time #h0
        kn_cat "story"))

(m318! (forall v22)
 (ant (p208 (act (m315 (action (m165 (lex bear))))) (agent v22)))
 (cq (p61 (class (m10 (lex mammal))) (member v22))))
(m317! (forall v22) (ant (p208))
 (cq (p210 (class (m18 (lex animal))) (member v22))))
(m316! (forall v22) (ant (p208))
 (cq (p209 (class (m37 (lex phys obj))) (member v22))))
(m314! (class (m144 (lex action))) (member (m212 (lex leap onto))))
(m302! (location (m301 (object b17) (sp-rel (m213 (lex on))))) (object b3)
 (time (m299 (a3 b17) (skf time-at))))
(m300! (after (m299)) (before b18))
(m298! (act (m294 (action (m212)) (object b17))) (agent b3) (time b18))
(m297! (act (m296 (action (m212)))) (agent b3))
(m295! (act (m294)) (agent b3) (kn_cat story) (time b18))
(m293! (class (m209 (lex breast))) (member (m208 (skf breast-of))))
(m292! (act (m286 (action (m158 (lex eat))))) (agent b12))
(m291! (act (m286)) (agent b15))
(m290! (act (m163 (action (m158)) (object (m162 (lex plant))))) (agent b12))
(m289! (act (m163)) (agent b15))
(m288! (class (m144)) (member (m158)))
(m287! (act (m286)) (agent b1))
(m285! (act (m160 (action (m158)) (object (m159 (lex meat))))) (agent b1))
(m284! (object b15) (rel (m82 (lex horse))))
(m283! (object b12) (rel (m193 (lex pony))))
(m282! (object b4) (rel (m49 (lex sword))))
(m281! (object (m39 (lex Round Table))) (rel (m40 (lex table))))
(m280! (object b1) (rel (m8 (lex dog))))
(m279! (class (m37)) (member b1))
(m278! (class (m35 (lex vertebrate))) (member b1))
(m277! (class (m18)) (member b1))
(m276! (class (m16 (lex predator))) (member b1))
(m275! (class (m14 (lex carnivore))) (member b1))
(m274! (class (m12 (lex quadruped))) (member b1))
(m273! (class (m10)) (member b1))
(m272! (class (m8)) (member b1))
(m271! (class (m24 (lex person))) (member b2))
(m270! (class (m77 (lex furniture))) (member (m39)))
```

```
(m269! (class (m40)) (member (m39)))
(m268! (class (m44 (lex king))) (member b3))
(m267! (class (m24)) (member b3))
(m266! (class (m49)) (member b4))
(m265! (class (m54 (lex wizard))) (member b5))
(m264! (class (m24)) (member b5))
(m263! (class (m44)) (member b6))
(m262! (class (m24)) (member b6))
(m261! (class (m44)) (member b7))
(m260! (class (m24)) (member b7))
(m259! (class (m44)) (member b8))
(m258! (class (m24)) (member b8))
(m257! (class (m68 (lex knight))) (member b9))
(m256! (class (m24)) (member b9))
(m255! (class (m68)) (member b10))
(m254! (class (m24)) (member b10))
(m253! (class (m68)) (member b11))
(m252! (class (m24)) (member b11))
(m251! (class (m89 (lex basic ctgy))) (member (m24)))
(m250! (class (m89)) (member (m8)))
(m249! (class (m89)) (member (m82)))
(m248! (class (m89)) (member (m2 (lex deer))))
(m247! (class (m89)) (member (m80 (lex chair))))
(m246! (class (m89)) (member (m40)))
(m245! (class (m97 (lex color))) (member (m96 (lex white))))
(m244! (class (m97)) (member (m99 (lex black))))
(m243! (class (m102 (lex size))) (member (m101 (lex small))))
(m242! (class (m102)) (member (m106 (lex large))))
(m241! (class (m111 (lex value))) (member (m110 (lex good))))
(m240! (class (m111)) (member (m113 (lex bad))))
(m239! (class (m193)) (member b12))
(m238! (class (m118 (lex equine))) (member b12))
(m237! (class (m37)) (member b12))
(m236! (class (m35)) (member b12))
(m235! (class (m28 (lex herbivore))) (member b12))
(m234! (class (m18)) (member b12))
(m233! (class (m12)) (member b12))
(m232! (class (m10)) (member b12))
(m231! (class (m24)) (member b14))
(m230! (class (m118)) (member b15))
(m229! (class (m82)) (member b15))
(m228! (class (m37)) (member b15))
(m227! (class (m35)) (member b15))
(m226! (class (m28)) (member b15))
(m225! (class (m18)) (member b15))
(m224! (class (m12)) (member b15))
(m223! (class (m10)) (member b15))
(m222! (class (m24)) (member b16))
(m210! (object (m208)) (rel (m209)))
(m132! (class (m24)))

(m318! m317! m316! m314! m302! m300! m298! m297! m295! m293! m292! m291! m290!
 m289! m288! m287! m285! m284! m283! m282! m281! m280! m279! m278! m277! m276!
 m275! m274! m273! m272! m271! m270! m269! m268! m267! m266! m265! m264! m263!
 m262! m261! m260! m259! m258! m257! m256! m255! m254! m253! m252! m251! m250!
 m249! m248! m247! m246! m245! m244! m243! m242! m241! m240! m239! m238! m237!
 m236! m235! m234! m233! m232! m231! m230! m229! m228! m227! m226! m225! m224!
 m223! m222! m210! m132!)
```

```
 CPU time : 19.08

*
;;; The thing he leaped onto is a hackney
(describe
(assert member *hack1
     class (build lex "hackney")
     kn_cat "story"))

(m382! (class (m381 (lex hackney))) (kn_cat story) (member b17))

(m382!)

 CPU time : 0.01

*
;;; The hackney is small
(describe
(assert object *hack1
        property (build lex "small")
        kn_cat "story"))

(m383! (kn_cat story) (object b17) (property (m101 (lex small))))

(m383!)

 CPU time : 0.00

*
;;; Hackney is an unknown word
(describe
(assert object (build lex "hackney")
        property (build lex "unknown")))

(m384! (object (m381 (lex hackney))) (property (m220 (lex unknown))))

(m384!)

 CPU time : 0.01

*
;;; A king leaps on a small hackney
(describe
(assert agent #king1
        act (build action (build lex "leap onto")
                  object #hack2)
        time *h0
        kn_cat "story"))

(m386! (act (m385 (action (m212 (lex leap onto))) (object b20))) (agent b19)
 (kn_cat story) (time b18))

(m386!)

 CPU time : 0.02

*
```

```
(describe
(assert object *hack2
        property (build lex "small")
        kn_cat "story"))

(m387! (kn_cat story) (object b20) (property (m101 (lex small))))

(m387!)

 CPU time : 0.02

*
(describe
(assert member *king1
        class (build lex "king")
        kn_cat "story"))

(m388! (class (m44 (lex king))) (kn_cat story) (member b19))

(m388!)

 CPU time : 0.01

*
(describe
(assert member *hack2
        class (build lex "hackney")
        kn_cat "story"))

(m389! (class (m381 (lex hackney))) (kn_cat story) (member b20))

(m389!)

 CPU time : 0.01

*
;;; A(nother) king leaps on a small hackney

(describe
(assert agent #king2
        act (build action (build lex "leap onto")
                  object #hack3)
        time *h0
        kn_cat "story"))

(m391! (act (m390 (action (m212 (lex leap onto))) (object b22))) (agent b21)
 (kn_cat story) (time b18))

(m391!)

 CPU time : 0.01

*
(describe
(assert member *king2
        class (build lex "king")
        kn_cat "story"))
```

```
(m392! (class (m44 (lex king))) (kn_cat story) (member b21))

(m392!)

 CPU time : 0.02

*
(describe
(assert member *hack3
        class (build lex "hackney")
        kn_cat "story"))

(m393! (class (m381 (lex hackney))) (kn_cat story) (member b22))

(m393!)

 CPU time : 0.01

*
(describe
(assert object *hack3
        property (build lex "small")
        kn_cat "story"))

(m394! (kn_cat story) (object b22) (property (m101 (lex small))))

(m394!)

 CPU time : 0.01

*
;;; What is a hackney?

^(
-->defineNoun 'hackney)
 Definition of hackney:
 Actions performed on a hackney: king leap onto,
 Possible Properties: small,
nil

 CPU time : 8.50

*
;;; [...] Sir Persaunt mounted an ambling hackney, conveyed them
;;; on their way and then commended them to G-d. [p 188]

;;; Sir Persaunt mounted an ambling hackney

(describe
(assert agent #Persaunt
     act (build action (build lex "mount")
               object #hack4)
     kn_cat "story"))

(m453! (act (m452 (action (m187 (lex mount))) (object b24))) (agent b23)
 (kn_cat story))

(m453!)
```

```
 CPU time : 0.01

*
(describe
(assert object *Persaunt
        proper-name (build lex "Sir Persaunt")
        kn_cat "story"))

(m455! (kn_cat story) (object b23) (proper-name (m454 (lex Sir Persaunt))))

(m455!)

 CPU time : 0.01

*
(describe
(assert member *Persaunt

        class (build lex "knight")
        kn_cat "story-comp"))

(m456! (class (m68 (lex knight))) (kn_cat story-comp) (member b23))

(m456!)

 CPU time : 0.02

*
(describe
(assert member *hack4
     class (build lex "hackney")
     kn_cat "story"))

(m457! (class (m381 (lex hackney))) (kn_cat story) (member b24))

(m457!)

 CPU time : 0.02

*
(describe
(assert object *hack4
     property (build lex "ambling")
     kn_cat "story"))

(m458! (kn_cat story) (object b24) (property (m127 (lex ambling))))

(m458!)

 CPU time : 0.00

*
(describe
(assert equiv (build skf "rider-of"
                  a2 *hack4)
     equiv *Persaunt
     kn_cat "story-comp"))
```

```
(m460! (equiv (m459 (a2 b24) (skf rider-of)) b23) (kn_cat story-comp))

(m460!)

 CPU time : 0.01

*
;;; What is a hackney?

^(
-->defineNoun 'hackney)
 Definition of hackney:
 Actions performed on a hackney: knight mount, king leap onto,
 Possible Properties: ambling, small,
nil

 CPU time : 2.79

*
;;; So the king took a little hackney and [...] came unto Sir Tristram's
;;; pavilion. When Sir Tristram saw the king he ran to him and would have
;;; held his stirrup, but the king leapt from his horse quickly [...] [p. 248]

;;; King Arthur took a little hackney

(describe
(assert agent *KA
        act (build action (build lex "take")
                  object #hack5)
        time #h1 kn_cat "story"))

(m482! (act (m481 (action (m480 (lex take))) (object b25))) (agent b3)
 (kn_cat story) (time b26))

(m482!)

 CPU time : 0.01

*
(describe
(assert before *h0
        after *h1))

(m483! (after b26) (before b18))

(m483!)

 CPU time : 0.01

*
(describe
(assert member *hack5
        class (build lex "hackney")
        kn_cat "story"))

(m484! (class (m381 (lex hackney))) (kn_cat story) (member b25))
```

```
(m484!)

 CPU time : 0.01

*
(describe
(assert object *hack5
        property (build lex "little")
        kn_cat "story"))

(m485! (kn_cat story) (object b25) (property (m104 (lex little))))

(m485!)

 CPU time : 0.02

*
;;; King Arthur arrived at Sir Tristram's pavilion.

(describe
(assert agent *KA
        act (build action (build lex "arrive")
                  place #pavilion2
                  time #h2)
        kn_cat "story"))

(m488! (act (m487 (action (m486 (lex arrive))) (place b27) (time b28)))
 (agent b3) (kn_cat story))

(m488!)

 CPU time : 0.02

*
(describe
(assert member *pavilion2
        class (build lex "pavilion")
        kn_cat "story"))

(m489! (class (m182 (lex pavilion))) (kn_cat story) (member b27))

(m489!)

 CPU time : 0.01

*
(describe
(assert object *pavilion2
        rel (build lex "pavilion")
        possessor *Tris
        kn_cat "story"))

(m490! (kn_cat story) (object b27) (possessor b11) (rel (m182 (lex pavilion))))

(m490!)

 CPU time : 0.01
```

```
*
(describe
(assert before *h1
        after *h2
        kn_cat "story-comp"))

(m491! (after b28) (before b26) (kn_cat story-comp))

(m491!)

 CPU time : 0.02

*
;;; Sir Tristram tried to hold King Arthur's stirrup, but King Arthur
;;; leaped from his horse to the ground quickly.

(describe
(assert agent *Tris
        act (build action (build lex "try")
                  object (build agent *Tris
                                act (build action (build lex "hold")
                                          object #stirrup1)))
        time *h2
        kn_cat "story"))

(m497!
 (act (m496 (action (m492 (lex try)))
       (object
        (m495 (act (m494 (action (m493 (lex hold))) (object b29)))
         (agent b11)))))
 (agent b11) (kn_cat story) (time b28))

(m497!)

 CPU time : 0.02

*
(describe
(assert member *stirrup1
        class (build lex "stirrup")
        kn_cat "story"))

(m499! (class (m498 (lex stirrup))) (kn_cat story) (member b29))

(m499!)

 CPU time : 0.02

*
(describe
        (assert object *stirrup1
        rel (build lex "stirrup")
        possessor *KA
        kn_cat "story"))

(m500! (kn_cat story) (object b29) (possessor b3) (rel (m498 (lex stirrup))))

(m500!)
```

```
 CPU time : 0.01

*
(describe
(assert min 0
        max 0
        arg (build agent *Tris
                   act (build action (build lex "hold")
                             object *stirrup1))
        kn_cat "story-comp"))

(m501! (min 0) (max 0)
 (arg (m495 (act (m494 (action (m493 (lex hold))) (object b29))) (agent b11)))
 (kn_cat story-comp))

(m501!)

 CPU time : 0.01

*
(clear-infer)

(Node activation cleared. Some register information retained.)

 CPU time : 0.33

*
;; King Arthur leaps from his horse
(describe
(add    agent *KA
        act (build action (build lex "leap from")
                   object #horsx)
        time *h2
        kn_cat "story"))
8528736 bytes have been tenured, next gc will be global.
See the documentation for variable excl:*global-gc-behavior* for more information.

(m707! (class (m144 (lex action))) (member (m116 (lex ride))))
(m706! (act (m703 (action (m116)))) (agent (m459 (a2 b24) (skf rider-of))))
(m705! (act (m701 (action (m116)) (object b24))) (agent (m459)))
(m704! (act (m703)) (agent b23))
(m702! (act (m701)) (agent b23))
(m700! (after (m699 (a3 b20) (skf time-at))) (before b18))
(m698! (equiv (m459) b23))
(m688! (min 0) (max 0)
 (arg (m687 (act (m686 (action (m493 (lex hold))))) (agent b11))))
(m685! (min 0) (max 0)
 (arg (m495 (act (m494 (action (m493)) (object b29))) (agent b11))))
(m684! (class (m37 (lex phys obj))) (member b24))
(m683! (class (m18 (lex animal))) (member b24))
(m652! (class (m144)) (member (m492 (lex try))))
(m651! (act (m650 (action (m492)))) (agent b11))
(m649! (class (m144)) (member (m486 (lex arrive))))
(m648! (act (m647 (action (m486)))) (agent b3))
(m646! (class (m144)) (member (m480 (lex take))))
(m645! (act (m644 (action (m480)))) (agent b3))
(m643! (class (m144)) (member (m187 (lex mount))))
```

```
(m642! (act (m641 (action (m187)))) (agent b23))
(m640! (act (m385 (action (m212 (lex leap onto))) (object b20))) (agent b19)
 (time b18))
(m639! (object b29) (rel (m498 (lex stirrup))))
(m638! (object b27) (rel (m182 (lex pavilion))))
(m602! (object b25) (property (m104 (lex little))))
(m601! (object b24) (property (m127 (lex ambling))))
(m600! (object b22) (property (m101 (lex small))))
(m599! (object b20) (property (m101)))
(m598! (object b17) (property (m101)))
(m556! (class (m44 (lex king))) (member b19))
(m555! (class (m24 (lex person))) (member b19))
(m554! (class (m44)) (member b21))
(m553! (class (m24)) (member b21))
(m552! (class (m381 (lex hackney))) (member b30))
(m551! (class (m68 (lex knight))) (member (m459)))
(m550! (class (m24)) (member (m459)))
(m549! (class (m68)) (member b23))
(m548! (class (m24)) (member b23))
(m547! (class (m381)) (member b24))
(m546! (class (m381)) (member b25))
(m545! (class (m183 (lex tent))) (member b27))
(m544! (class (m182)) (member b27))
(m543! (class (m498)) (member b29))
(m526! (equiv b30 b17))
(m525! (class (m144)) (member (m217 (lex leap from))))
(m512! (after b28) (before b26))
(m511! (after b28) (before b18))
(m510! (location (m509 (object b30) (sp-rel (m213 (lex on)))) (object b3)
 (time (m507 (a4 b30) (skf time-at))))
(m508! (after b28) (before (m507)))
(m506! (act (m502 (action (m217)) (object b30))) (agent b3) (time b28))
(m505! (act (m504 (action (m217)))) (agent b3))
(m503! (act (m502)) (agent b3) (kn_cat story) (time b28))
(m483! (after b26) (before b18))
(m451! (after (m450 (a3 b22) (skf time-at))) (before b18))
(m449! (act (m390 (action (m212)) (object b22))) (agent b21) (time b18))
(m401! (act (m296 (action (m212)))) (agent b21))
(m400! (act (m296)) (agent b19))
(m399! (class (m381)) (member b17))
(m398! (class (m381)) (member b20))
(m397! (class (m381)) (member b22))
(m384! (object (m381)) (property (m220 (lex unknown))))
(m316! (forall v22)
 (ant (p208 (act (m315 (action (m165 (lex bear))))) (agent v22)))
 (cq (p209 (class (m37)) (member v22))))
(m314! (class (m144)) (member (m212)))
(m300! (after (m299 (a3 b17) (skf time-at))) (before b18))
(m298! (act (m294 (action (m212)) (object b17))) (agent b3) (time b18))
(m297! (act (m296)) (agent b3))
(m293! (class (m209 (lex breast))) (member (m208 (skf breast-of))))
(m292! (act (m286 (action (m158 (lex eat))))) (agent b12))
(m291! (act (m286)) (agent b15))
(m290! (act (m163 (action (m158)) (object (m162 (lex plant))))) (agent b12))
(m289! (act (m163)) (agent b15))
(m288! (class (m144)) (member (m158)))
(m287! (act (m286)) (agent b1))
(m285! (act (m160 (action (m158)) (object (m159 (lex meat))))) (agent b1))
```

```
(m284! (object b15) (rel (m82 (lex horse))))
(m283! (object b12) (rel (m193 (lex pony))))
(m282! (object b4) (rel (m49 (lex sword))))
(m281! (object (m39 (lex Round Table))) (rel (m40 (lex table))))
(m280! (object b1) (rel (m8 (lex dog))))
(m279! (class (m37)) (member b1))
(m278! (class (m35 (lex vertebrate))) (member b1))
(m277! (class (m18)) (member b1))
(m276! (class (m16 (lex predator))) (member b1))
(m275! (class (m14 (lex carnivore))) (member b1))
(m274! (class (m12 (lex quadruped))) (member b1))
(m273! (class (m10 (lex mammal))) (member b1))
(m272! (class (m8)) (member b1))
(m271! (class (m24)) (member b2))
(m270! (class (m77 (lex furniture))) (member (m39)))
(m269! (class (m40)) (member (m39)))
(m268! (class (m44)) (member b3))
(m267! (class (m24)) (member b3))
(m266! (class (m49)) (member b4))
(m265! (class (m54 (lex wizard))) (member b5))
(m264! (class (m24)) (member b5))
(m263! (class (m44)) (member b6))
(m262! (class (m24)) (member b6))
(m261! (class (m44)) (member b7))
(m260! (class (m24)) (member b7))
(m259! (class (m44)) (member b8))
(m258! (class (m24)) (member b8))
(m257! (class (m68)) (member b9))
(m256! (class (m24)) (member b9))
(m255! (class (m68)) (member b10))
(m254! (class (m24)) (member b10))
(m253! (class (m68)) (member b11))
(m252! (class (m24)) (member b11))
(m251! (class (m89 (lex basic ctgy))) (member (m24)))
(m250! (class (m89)) (member (m8)))
(m249! (class (m89)) (member (m82)))
(m248! (class (m89)) (member (m2 (lex deer))))
(m247! (class (m89)) (member (m80 (lex chair))))
(m246! (class (m89)) (member (m40)))
(m245! (class (m97 (lex color))) (member (m96 (lex white))))
(m244! (class (m97)) (member (m99 (lex black))))
(m243! (class (m102 (lex size))) (member (m101)))
(m242! (class (m102)) (member (m106 (lex large))))
(m241! (class (m111 (lex value))) (member (m110 (lex good))))
(m240! (class (m111)) (member (m113 (lex bad))))
(m239! (class (m193)) (member b12))
(m238! (class (m118 (lex equine))) (member b12))
(m237! (class (m37)) (member b12))
(m236! (class (m35)) (member b12))
(m235! (class (m28 (lex herbivore))) (member b12))
(m234! (class (m18)) (member b12))
(m233! (class (m12)) (member b12))
(m232! (class (m10)) (member b12))
(m231! (class (m24)) (member b14))
(m230! (class (m118)) (member b15))
(m229! (class (m82)) (member b15))
(m228! (class (m37)) (member b15))
(m227! (class (m35)) (member b15))
```

```
(m226! (class (m28)) (member b15))
(m225! (class (m18)) (member b15))
(m224! (class (m12)) (member b15))
(m223! (class (m10)) (member b15))
(m222! (class (m24)) (member b16))
(m210! (object (m208)) (rel (m209)))

(m707! m706! m705! m704! m702! m700! m698! m688! m685! m684! m683! m652! m651!
 m649! m648! m646! m645! m643! m642! m640! m639! m638! m602! m601! m600! m599!
 m598! m556! m555! m554! m553! m552! m551! m550! m549! m548! m547! m546! m545!
 m544! m543! m526! m525! m512! m511! m510! m508! m506! m505! m503! m483! m451!
 m449! m401! m400! m399! m398! m397! m384! m316! m314! m300! m298! m297! m293!
 m292! m291! m290! m289! m288! m287! m285! m284! m283! m282! m281! m280! m279!
 m278! m277! m276! m275! m274! m273! m272! m271! m270! m269! m268! m267! m266!
 m265! m264! m263! m262! m261! m260! m259! m258! m257! m256! m255! m254! m253!
 m252! m251! m250! m249! m248! m247! m246! m245! m244! m243! m242! m241! m240!
 m239! m238! m237! m236! m235! m234! m233! m232! m231! m230! m229! m228! m227!
 m226! m225! m224! m223! m222! m210!)

 CPU time : 47.59

*
;; King Arthur leaps onto the ground
(describe
(add agent *KA
     act (build action (build lex "leap onto")
                object (build lex "ground"))
     time *h2
     kn_cat "story"))

(m760! (after (m759 (a3 (m755 (lex ground))) (skf time-at))) (before b28))
(m758! (act (m756 (action (m212 (lex leap onto))) (object (m755)))) (agent b3)
 (time b28))
(m757! (act (m756)) (agent b3) (kn_cat story) (time b28))
(m297! (act (m296 (action (m212)))) (agent b3))

(m760! m758! m757! m297!)

 CPU time : 0.40

*
(describe
(assert member *horsx
        class (build lex "horse")
        kn_cat "story"))

(m761! (class (m82 (lex horse))) (kn_cat story) (member b30))

(m761!)

 CPU time : 0.01

*
(describe
(assert object *horsx
        possessor *KA
        rel (build lex "horse")
        kn_cat "story"))
```

```
(m762! (kn_cat story) (object b30) (possessor b3) (rel (m82 (lex horse))))

(m762!)

 CPU time : 0.00

*
(describe
(assert equiv (build skf "rider-of"
                     a2 *horsx)
        equiv *KA kn_cat "story-comp"))

(m764! (equiv (m763 (a2 b30) (skf rider-of)) b3) (kn_cat story-comp))

(m764!)

 CPU time : 0.01

*
(describe
(assert cause (build agent *KA
               act (build action (build lex "leap onto")
                          object (build lex "ground"))
               time *h2)
        effect (build min 0
                      max 0
                      arg (build agent *Tris
                                 act (build action (build lex "hold")
                                            object *stirrup1)))
        kn_cat "story-comp"))

(m765!
 (cause
  (m758!
   (act (m756 (action (m212 (lex leap onto))) (object (m755 (lex ground)))))
   (agent b3) (time b28)))
 (effect
  (m685! (min 0) (max 0)
   (arg
    (m495 (act (m494 (action (m493 (lex hold))) (object b29))) (agent b11)))))
 (kn_cat story-comp))

(m765!)

 CPU time : 0.02

*
;;; What is a hackney?

^(
-->defineNoun 'hackney)
 Definition of hackney:
 Possible Class Inclusions: mammal, horse,
 Possible Actions: eat plant,
 Possible Properties: ambling, little, small,
 Possessive: person horse, king horse,
nil
```

```
 CPU time : 37.48

*
;;; Then the old knight took a little hackney and rode for Sir Palomides
;;; and brought him to his own manor. [p. 327]

;;; The old knight took a little hackney.

(describe
(assert agent #hkn1
        act (build action (build lex "take")
                  object #hack6)
        time #h3
        kn_cat "story"))

(m846! (act (m845 (action (m480 (lex take))) (object b32))) (agent b31)
 (kn_cat story) (time b33))

(m846!)

 CPU time : 0.00

*
(describe
(assert member *hkn1

        class (build lex "knight")
        kn_cat "story"))

(m847! (class (m68 (lex knight))) (kn_cat story) (member b31))

(m847!)

 CPU time : 0.02

*
(describe
(assert object *hkn1
        property (build lex "old")
        kn_cat "story"))

(m848! (kn_cat story) (object b31) (property (m178 (lex old))))

(m848!)

 CPU time : 0.01

*
(describe
(assert object *hack6
        property (build lex "little")
        kn_cat "story"))

(m849! (kn_cat story) (object b32) (property (m104 (lex little))))

(m849!)
```

```
 CPU time : 0.00

*
(describe
(assert member *hack6

        class (build lex "hackney")
        kn_cat "story"))

(m850! (class (m381 (lex hackney))) (kn_cat story) (member b32))

(m850!)

 CPU time : 0.02

*
;;; The old knight rode to Sir Palomides.

(describe
(assert agent *hkn1
     act (build action (build lex "ride")
                to #Palomides
                time #h4)
     kn_cat "story"))

(m852! (act (m851 (action (m116 (lex ride))) (time b35) (to b34))) (agent b31)
 (kn_cat story))

(m852!)

 CPU time : 0.01

*
(describe
(assert equiv (build skf "rider-of"
                   a2 *hack6)
     equiv *hkn1
     kn_cat "story-comp"))

(m854! (equiv (m853 (a2 b32) (skf rider-of)) b31) (kn_cat story-comp))

(m854!)

 CPU time : 0.01

*
(describe
(assert before *h2
        after *h3
        kn_cat "story-comp"))

(m855! (after b33) (before b28) (kn_cat story-comp))

(m855!)

 CPU time : 0.00

*
```

```
(describe
(assert before *h3
        after *h4
        kn_cat "story-comp"))

(m856! (after b35) (before b33) (kn_cat story-comp))

(m856!)

 CPU time : 0.01

*
(describe
(assert object *Palomides
        proper-name (build lex "Sir Palomides")
        kn_cat "story"))

(m858! (kn_cat story) (object b34) (proper-name (m857 (lex Sir Palomides))))

(m858!)

 CPU time : 0.01

*
(describe
(assert member *Palomides

        class (build lex "knight")
        kn_cat "story-comp"))

(m859! (class (m68 (lex knight))) (kn_cat story-comp) (member b34))

(m859!)

 CPU time : 0.01

*
;;; What is a hackney?

^(
-->defineNoun 'hackney)
 Definition of hackney:
 Possible Class Inclusions: horse,
 Possible Actions: eat plant,
 Possible Properties: ambling, little, small,
 Possessive: person horse, king horse,
nil

 CPU time : 299.50

*
;;; As soon as the squire heard these words, he alighted from his hackney
;;; and kneeled down at Galahad's feet, and prayed him that he might go with
;;; him til he had made him a knight. [p. 335]

;;; The squire alighed from his hackney.

(describe
```

```
(assert agent #squire1
     act (build action (build lex "alight")
               from #hack7)
     kn_cat "story"))

(m944! (act (m943 (action (m942 (lex alight))) (from b37))) (agent b36)
 (kn_cat story))

(m944!)

 CPU time : 0.03

*
(describe
(assert member *squire1
     class (build lex "squire")
     kn_cat "story"))

(m945! (class (m154 (lex squire))) (kn_cat story) (member b36))

(m945!)

 CPU time : 0.02

*
(describe
(assert member *hack7

     class (build lex "hackney")
     kn_cat "story"))

(m946! (class (m381 (lex hackney))) (kn_cat story) (member b37))

(m946!)

 CPU time : 0.02

*
(describe
(assert object *hack7
     possessor *squire1
     rel (build lex "hackney")
     kn_cat "story"))

(m947! (kn_cat story) (object b37) (possessor b36) (rel (m381 (lex hackney))))

(m947!)

 CPU time : 0.01

*
;;; What is a hackney?

^(
-->defineNoun 'hackney)
 Definition of hackney:
 Class Inclusions: horse,
 Possible Actions: eat plant,
```

```
 Possible Properties: ambling, little, small,
 Possessive: person horse, king horse, squire,
 Possibly Similar Items: pony,
nil

 CPU time : 64.87

*
;;; [Sir Percival] met a yeoman riding upon a hackney who led in his hand
;;; a great steed, blacker than any berry.

(describe
(assert agent #Perci
        act (build action (build lex "meet")
                  object #hyeo1)
        time #h5
        kn_cat "story"))

(m986! (act (m985 (action (m984 (lex meet))) (object b39))) (agent b38)
 (kn_cat story) (time b40))

(m986!)

 CPU time : 0.05

*
(describe
(assert member *Perci

        class (build lex "knight")
        kn_cat "story-comp"))

(m987! (class (m68 (lex knight))) (kn_cat story-comp) (member b38))

(m987!)

 CPU time : 0.01

*
(describe
(assert object *Perci
        proper-name (build lex "Sir Percival")
        kn_cat "story"))

(m989! (kn_cat story) (object b38) (proper-name (m988 (lex Sir Percival))))

(m989!)

 CPU time : 0.02

*
(describe
(assert member *hyeo1

        class (build lex "yeoman")
        kn_cat "story"))

(m990! (class (m156 (lex yeoman))) (kn_cat story) (member b39))
```

```
(m990!)

 CPU time : 0.01

*
(describe (assert before *h4
                  after *h5
                  kn_cat "story-comp"))

(m991! (after b40) (before b35) (kn_cat story-comp))

(m991!)

 CPU time : 0.02

*
;;; The yeoman was riding a hackney

(describe
(assert agent *hyeo1
        act (build action (build lex "ride")
                  object #hack8)
        time *h5
        kn_cat "story"))

(m993! (act (m992 (action (m116 (lex ride))) (object b41))) (agent b39)
 (kn_cat story) (time b40))

(m993!)

 CPU time : 0.02

*
(describe
(assert member *hack8
        class (build lex "hackney")
        kn_cat "story"))

(m994! (class (m381 (lex hackney))) (kn_cat story) (member b41))

(m994!)

 CPU time : 0.02

*
;;; The yeoman was leading a great black steed.

(describe
(assert agent *hyeo1
        act (build action (build lex "lead")
                  object #blksteed1)
        time *h5
        kn_cat "story"))

(m997! (act (m996 (action (m995 (lex lead))) (object b42))) (agent b39)
 (kn_cat story) (time b40))
```

```
(m997!)

 CPU time : 0.03

*
(describe
(assert member *blksteed1

        class (build lex "steed")
        kn_cat "story"))

(m999! (class (m998 (lex steed))) (kn_cat story) (member b42))

(m999!)

 CPU time : 0.02

*
(describe
(assert object *blksteed1
        property (build lex "great")
        property (build lex "black")
        kn_cat "story"))

(m1001! (kn_cat story) (object b42)
 (property (m1000 (lex great)) (m99 (lex black))))

(m1001!)

 CPU time : 0.02

*
;;; [A knight stole the steed]

(describe
(assert agent #thf_knt
        act (build action (build lex "steal")
                  object *blksteed1)
        time #h6
        kn_cat "story"))

(m1004! (act (m1003 (action (m1002 (lex steal))) (object b42))) (agent b43)
 (kn_cat story) (time b44))

(m1004!)

 CPU time : 0.02

*
(describe (assert before *h5
                  after *h6
                  kn_cat "story-comp"))

(m1005! (after b44) (before b40) (kn_cat story-comp))

(m1005!)

 CPU time : 0.02
```

```
*
(describe
(assert member *thf_knt
        class (build lex "knight")
        kn_cat "story"))

(m1006! (class (m68 (lex knight))) (kn_cat story) (member b43))

(m1006!)

 CPU time : 0.00

*
;;; 'Well,' said Sir Percival, 'what wouldst thou that I did? Thou seest
;;; well that I am on foot, but if I had a good horse I should soon bring
;;; him back.' ;Sir,' said the yeoman, 'take my hackney and do the best ye
;;; can, and I shall follow you on foot to know how ye shall speed.'

(describe
(assert agent *hyeo1
        act (build action (build lex "address")
                  object *Perci)
        time #h7
        kn_cat "story-comp"))

(m1009! (act (m1008 (action (m1007 (lex address))) (object b38))) (agent b39)
 (kn_cat story-comp) (time b45))

(m1009!)

 CPU time : 0.04

*
;;; The yeoman asked that Sir Percival take the hackney and that he try
;;; to catch the knight.

(describe
(assert agent *hyeo1
        act (build action (build lex "ask that")
                  object (build agent *Perci
                                act (build action (build lex "take")
                                          object *hack8))
                  object (build agent *Perci
                                act (build action (build lex "try")
                                          object (build agent *Perci
                                                        act (build action
                                                              (build lex "catch")
                                                              object *thf_knt )))))
        time *h7
        kn_cat "story"))

(m1019!
 (act (m1018 (action (m1010 (lex ask that)))
       (object
        (m1017
         (act (m1016 (action (m492 (lex try)))
               (object
```

```
                (m1015 (act (m1014 (action (m1013 (lex catch))) (object b43)))
                 (agent b38)))))
         (agent b38))
        (m1012 (act (m1011 (action (m480 (lex take))) (object b41)))
         (agent b38)))))
 (agent b39) (kn_cat story) (time b45))

(m1019!)

 CPU time : 0.07

*
(describe
(assert before *h6
        after *h7
        kn_cat "story-comp"))

(m1020! (after b45) (before b44) (kn_cat story-comp))

(m1020!)

 CPU time : 0.02

*
;;; Then Sir Percival bestrode the hackney and rode as fast as he might,
;;; and at last he saw the knight.  Then he cried,'Knight, turn back!'
;;; He turned and set his spear toward Sir Percival, and smote the hackney
;;; in the middle of the breast, so that it fell down dead to the earth.
;;; [pp. 552 - 553]

;;; Sir Percival bestrode the hackney

(describe
(assert agent *Perci
        act (build action (build lex "bestride")
                  object *hack8)
        time #h8
        kn_cat "story"))

(m1023! (act (m1022 (action (m1021 (lex bestride))) (object b41))) (agent b38)
 (kn_cat story) (time b46))

(m1023!)

 CPU time : 0.03

*
(describe
(assert before *h7
        after *h8
        kn_cat "story-comp"))

(m1024! (after b46) (before b45) (kn_cat story-comp))

(m1024!)

 CPU time : 0.01
```

```
*
(describe
(assert agent *Perci
     act (build action (build lex "ride")
               object *hack8)
     manner (build lex "fast")
     time *h8
     kn_cat "story"))

(m1026! (act (m992 (action (m116 (lex ride))) (object b41))) (agent b38)
 (kn_cat story) (manner (m1025 (lex fast))) (time b46))

(m1026!)

 CPU time : 0.02

*
;;; The thieving knight set his spear toward Sir Percival.

(describe
(assert agent *thf_knt
        act (build action (build lex "set")
                  object #aspear)
        direction *Perci
        time #h9
        kn_cat "story"))

(m1029! (act (m1028 (action (m1027 (lex set))) (object b47))) (agent b43)
 (direction b38) (kn_cat story) (time b48))

(m1029!)

 CPU time : 0.06

*
(describe
(assert before *h8
        after *h9
        kn_cat "story-comp"))

(m1030! (after b48) (before b46) (kn_cat story-comp))

(m1030!)

 CPU time : 0.02

*
(describe
(assert member *aspear
        class (build lex "spear")
        kn_cat "story"))

(m1031! (class (m147 (lex spear))) (kn_cat story) (member b47))

(m1031!)

 CPU time : 0.02
```

```
*
(describe
(assert object *aspear
        possessor *thf_knt
        rel (build lex "spear")
        kn_cat "story"))

(m1032! (kn_cat story) (object b47) (possessor b43) (rel (m147 (lex spear))))

(m1032!)

 CPU time : 0.02

*
;;; The knight smote the hackney in the breast

(describe
(assert agent *thf_knt
        act (build action (build lex "smite")
                  object *hack8)
        location #loc1
        time #h10
        kn_cat "story"))

(m1035! (act (m1034 (action (m1033 (lex smite))) (object b41))) (agent b43)
 (kn_cat story) (location b49) (time b50))

(m1035!)

 CPU time : 0.04

*
(describe
(assert member *loc1
        class (build lex "breast")
        kn_cat "story"))

(m1036! (class (m209 (lex breast))) (kn_cat story) (member b49))

(m1036!)

 CPU time : 0.02

*
(describe
(assert object *loc1
        possessor *hack8
        rel (build lex "breast")
        kn_cat "story"))

(m1037! (kn_cat story) (object b49) (possessor b41) (rel (m209 (lex breast))))

(m1037!)

 CPU time : 0.03

*
(describe
```

```
(assert part *loc1
        whole *hack8
        kn_cat "story-comp"))

(m1038! (kn_cat story-comp) (part b49) (whole b41))

(m1038!)

 CPU time : 0.02

*
(describe
(assert before *h9
        after *h10
        kn_cat "story-comp"))

(m1039! (after b50) (before b48) (kn_cat story-comp))

(m1039!)

 CPU time : 0.02

*
;;; The hackney fell to the earth

(describe
(assert object *hack8
        act (build action (build lex "fall")
                  to (build lex "earth")
                  time *h10)
        kn_cat "story"))

(m1043!
 (act (m1042 (action (m1040 (lex fall))) (time b50) (to (m1041 (lex earth)))))
 (kn_cat story) (object b41))

(m1043!)

 CPU time : 0.03

*
;;; It was dead

(describe
(assert object *hack8
        property (build lex "dead")
        time *h10
        kn_cat "story"))

(m1045! (kn_cat story) (object b41) (property (m1044 (lex dead))) (time b50))

(m1045!)

 CPU time : 0.03

*
;;; What is a hackney?
```

```
^(
-->defineNoun 'hackney)
 Definition of hackney:
 Class Inclusions: horse,
 Possible Structure: breast,
 Possible Actions: eat plant,
 Possible Properties: dead, ambling, little, small,
 Possessive: person horse, king horse, squire,
 Possibly Similar Items: pony,
nil

 CPU time : 51.51

*

End of /projects/stn2/CVA/demos/hackney.demo demonstration.

 CPU time : 542.44

*
```

B.4 Joust

```
Starting image '/util/acl62/composer'
  with no arguments
  in directory '/projects/stn2/CVA/'
  on machine 'localhost'.

International Allegro CL Enterprise Edition
6.2 [Solaris] (Aug 15, 2002 14:24)
Copyright (C) 1985-2002, Franz Inc., Berkeley, CA, USA.  All Rights Reserved.

This development copy of Allegro CL is licensed to:
   [4549] SUNY/Buffalo, N. Campus

;; Optimization settings: safety 1, space 1, speed 1, debug 2.
;; For a complete description of all compiler switches given the current
;; optimization settings evaluate (explain-compiler-settings).
;;---
;; Current reader case mode: :case-sensitive-lower
cl-user(1): :ld lna
; Loading /projects/stn2/CVA/lna.cl
;   Loading /projects/snwiz/bin/sneps.lisp
Loading system SNePS...10% 20% 30% 40% 50% 60% 70% 80% 90% 100%
SNePS-2.6 [PL:0a 2002/09/30 22:37:46] loaded.
Type '(sneps)' or '(snepslog)' to get started.
;   Loading /projects/stn2/CVA/defun_noun.cl
;   Loading /projects/stn2/CVA/defun_verb.cl

   Welcome to SNePS-2.6 [PL:0a 2002/09/30 22:37:46]

Copyright (C) 1984--2002 by Research Foundation of
State University of New York. SNePS comes with ABSOLUTELY NO WARRANTY!
Type '(copyright)' for detailed copyright information.
Type '(demo)' for a list of example applications.

   4/21/2003 10:49:52
```

```
* (demo "demos/joust.demo")

File /projects/stn2/CVA/demos/joust.demo is now the source of input.

 CPU time : 0.01

* ;; Joust Demo - originally created by K. Ehrlich
;;            - modified by S. Napieralski

;; clear all info
(resetnet t)

Net reset

 CPU time : 0.00

*
;; turn off singular path inference
^(
--> in-package snip)
#<The snip package>

 CPU time : 0.01

*
;;; redefine function to return nil
;;; so that forward inference will not be limited
^(
--> defun broadcast-one-report (rep)
   (let (anysent)
     (do.chset (ch *OUTGOING-CHANNELS* anysent)
               (when (isopen.ch ch)
                 (setq anysent (or (try-to-send-report rep ch) anysent)))))
   nil)
broadcast-one-report

 CPU time : 0.00

*
;;; return to sneps package
^(
--> in-package sneps)
#<The sneps package>

 CPU time : 0.00

*
^
--> (setf snepsul::*infertrace* nil)
nil

 CPU time : 0.01

*
;;; load background knowledge
(intext "/projects/stn2/CVA/demos/rels")
File /projects/stn2/CVA/demos/rels is now the source of input.
```

```
 CPU time : 0.00

*

(a1 a2 a3 a4 after agent against antonym associated before cause class
 direction equiv etime event from in indobj instr into lex location manner
 member mode object on onto part place possessor proper-name property rel skf
 sp-rel stime subclass superclass subset superset synonym time to whole kn_cat)

 CPU time : 0.04

*

End of file /projects/stn2/CVA/demos/rels

 CPU time : 0.05

* (demo "/projects/stn2/CVA/demos/paths")

File /projects/stn2/CVA/demos/paths is now the source of input.

 CPU time : 0.00

* ;;; Composition of certain paths

;;; Make Before Transitive
(define-path before (compose before (kstar (compose after- ! before))))
before implied by the path (compose before (kstar (compose after- ! before)))
before- implied by the path (compose (kstar (compose before- ! after)) before-)

 CPU time : 0.00

*
;;; Make After Transitive
(define-path after (compose after (kstar (compose before- ! after))))
after implied by the path (compose after (kstar (compose before- ! after)))
after- implied by the path (compose (kstar (compose after- ! before)) after-)

 CPU time : 0.00

*
;; If X is a member of the class Y and Y is a subclass of Z then
;;   X is a member of the class Z.
;; Example:  If Fido is a brachet and all brachets are hounds then
;;;  Fido is a hound.
(define-path class (compose class (kstar (compose subclass- ! superclass))))
class implied by the path (compose class
                           (kstar (compose subclass- ! superclass)))
class- implied by the path (compose (kstar (compose superclass- ! subclass))
                            class-)

 CPU time : 0.00

*
;; Make subclass transitive
(define-path subclass (compose subclass (kstar (compose superclass- ! subclass))))
subclass implied by the path (compose subclass
                              (kstar (compose superclass- ! subclass)))
```

```
subclass- implied by the path (compose
                               (kstar (compose subclass- ! superclass))
                               subclass-)

 CPU time : 0.00

*
;; Make member move over equiv arcs
(define-path member (compose member (kstar (compose equiv- ! equiv))))
member implied by the path (compose member (kstar (compose equiv- ! equiv)))
member- implied by the path (compose (kstar (compose equiv- ! equiv)) member-)

 CPU time : 0.00

*

End of /projects/stn2/CVA/demos/paths demonstration.

 CPU time : 0.02

* (demo "/projects/stn2/CVA/demos/joust.base")

File /projects/stn2/CVA/demos/joust.base is now the source of input.

 CPU time : 0.00

* ;;; This is the set of assertions which build the base network which
;;; corresponds to Karen Ehrlich's vocabulary acquisition project.
;;; Commented and made accessible by Alan Hunt and Geoffrey Koplas, '97
;;; The following is the information that needs to get fed into the network
;;; for the Narrative Acquisition demos; namely the background knowledge
;;; on the words _other_ than the one being acquired.

; Animals are physical objects
(describe
(assert subclass (build lex "animal") superclass (build lex "phys obj")
        kn_cat "life"))

(m3! (kn_cat life) (subclass (m1 (lex animal)))
 (superclass (m2 (lex phys obj))))

(m3!)

 CPU time : 0.00

*
; Quadrupeds are vertebrates
(describe
(assert subclass (build lex "quadruped") superclass (build lex "vertebrate")
        kn_cat "life"))

(m6! (kn_cat life) (subclass (m4 (lex quadruped)))
 (superclass (m5 (lex vertebrate))))

(m6!)

 CPU time : 0.00
```

```
*
; Ungulates are herbivores
(describe
(assert subclass (build lex "ungulate") superclass (build lex "herbivore")
        kn_cat "life"))

(m9! (kn_cat life) (subclass (m7 (lex ungulate)))
 (superclass (m8 (lex herbivore))))

(m9!)

 CPU time : 0.00

*
; Herbivore and carnivore are antonyms
(describe
(assert antonym (build lex "herbivore") antonym (build lex "carnivore")))

(m11! (antonym (m10 (lex carnivore)) (m8 (lex herbivore))))

(m11!)

 CPU time : 0.00

*
; Mammals are animals
(describe
(assert subclass (build lex "mammal") superclass (build lex "animal")
        kn_cat "life"))

(m13! (kn_cat life) (subclass (m12 (lex mammal)))
 (superclass (m1 (lex animal))))

(m13!)

 CPU time : 0.01

*
; Mammals are vertebrates
(describe
(assert subclass (build lex "mammal") superclass (build lex "vertebrate")
        kn_cat "life"))

(m14! (kn_cat life) (subclass (m12 (lex mammal)))
 (superclass (m5 (lex vertebrate))))

(m14!)

 CPU time : 0.01

*
; Deer are mammals
(describe
(assert subclass (build lex "deer") superclass (build lex "mammal")
        kn_cat "life"))

(m16! (kn_cat life) (subclass (m15 (lex deer)))
 (superclass (m12 (lex mammal))))
```

```
(m16!)

 CPU time : 0.00

*
; Deer are quadrupeds
(describe
(assert subclass (build lex "deer") superclass (build lex "quadruped")
        kn_cat "life"))

(m17! (kn_cat life) (subclass (m15 (lex deer)))
 (superclass (m4 (lex quadruped))))

(m17!)

 CPU time : 0.00

*
; Deer are herbivores
(describe
(assert subclass (build lex "deer") superclass (build lex "herbivore")
        kn_cat "life"))

(m18! (kn_cat life) (subclass (m15 (lex deer)))
 (superclass (m8 (lex herbivore))))

(m18!)

 CPU time : 0.00

*
; Deer are animals
(describe (assert subclass (build lex "deer") superclass (build lex "animal")
                  kn_cat "life"))

(m19! (kn_cat life) (subclass (m15 (lex deer))) (superclass (m1 (lex animal))))

(m19!)

 CPU time : 0.00

*
; Deer is a basic level category
(describe
(assert member (build lex "deer")  class (build lex "basic ctgy")
        kn_cat "life"))

(m21! (class (m20 (lex basic ctgy))) (kn_cat life) (member (m15 (lex deer))))

(m21!)

 CPU time : 0.00

*
; Horses are quadrupeds
(describe
(assert subclass (build lex "horse") superclass (build lex "quadruped")
```

```
        kn_cat "life"))

(m23! (kn_cat life) (subclass (m22 (lex horse)))
 (superclass (m4 (lex quadruped))))

(m23!)

 CPU time : 0.00

*
; Horses are herbivores
(describe
(assert subclass (build lex "horse") superclass (build lex "herbivore")
        kn_cat "life"))

(m24! (kn_cat life) (subclass (m22 (lex horse)))
 (superclass (m8 (lex herbivore))))

(m24!)

 CPU time : 0.00

*
; Horses are animals
(describe
(assert subclass (build lex "horse") superclass (build lex "animal")
        kn_cat "life"))

(m25! (kn_cat life) (subclass (m22 (lex horse)))
 (superclass (m1 (lex animal))))

(m25!)

 CPU time : 0.01

*
; Horse is a basic level category
(describe
(assert member (build lex "horse")  class (build lex "basic ctgy")
        kn_cat "life"))

(m26! (class (m20 (lex basic ctgy))) (kn_cat life) (member (m22 (lex horse))))

(m26!)

 CPU time : 0.00

*
; Ponies are animals
(describe (assert subclass (build lex "pony") superclass (build lex "animal")
                 kn_cat "life"))

(m28! (kn_cat life) (subclass (m27 (lex pony))) (superclass (m1 (lex animal))))

(m28!)

 CPU time : 0.01
```

```
*
;"Dogs are mammals"
(describe
(assert subclass (build lex "dog") superclass (build lex "mammal")
        kn_cat "life"))

(m30! (kn_cat life) (subclass (m29 (lex dog))) (superclass (m12 (lex mammal))))

(m30!)

 CPU time : 0.00

*
;"Dogs are quadrupeds"
(describe
(assert subclass (build lex "dog") superclass (build lex "quadruped")
        kn_cat "life"))

(m31! (kn_cat life) (subclass (m29 (lex dog)))
 (superclass (m4 (lex quadruped))))

(m31!)

 CPU time : 0.00

*
; "Dogs are carnivores"
(describe
(assert subclass (build lex "dog") superclass (build lex "carnivore")))

(m32! (subclass (m29 (lex dog))) (superclass (m10 (lex carnivore))))

(m32!)

 CPU time : 0.00

*
;"Dogs are animals"
(describe
(assert subclass (build lex "dog") superclass (build lex "animal")
        kn_cat "life"))

(m33! (kn_cat life) (subclass (m29 (lex dog))) (superclass (m1 (lex animal))))

(m33!)

 CPU time : 0.00

*
; Dog is a basic level category
(describe
 (assert member (build lex "dog")  class (build lex "basic ctgy")
        kn_cat "life"))

(m34! (class (m20 (lex basic ctgy))) (kn_cat life) (member (m29 (lex dog))))

(m34!)
```

```
 CPU time : 0.00

*
;"Hounds are dogs"
(describe
(assert subclass (build lex "hound") superclass (build lex "dog")
        kn_cat "life"))

(m36! (kn_cat life) (subclass (m35 (lex hound))) (superclass (m29 (lex dog))))

(m36!)

 CPU time : 0.00

*
;"Something is a member of the class dog"
(describe
(assert member #rex class (build lex "dog")))

(m37! (class (m29 (lex dog))) (member b1))

(m37!)

 CPU time : 0.00

*
;"The dog is possesed by something"
(describe
(assert object *rex rel (build lex "dog") possessor #rexboss))

(m38! (object b1) (possessor b2) (rel (m29 (lex dog))))

(m38!)

 CPU time : 0.00

*
;"The thing that owns the dog is a person"
(describe
(assert member *rexboss class (build lex "person")))

(m40! (class (m39 (lex person))) (member b2))

(m40!)

 CPU time : 0.00

*
; Kings are persons
(describe
(add subclass (build lex "king") superclass (build lex "person")
        kn_cat "life"))

(m42! (kn_cat life) (subclass (m41 (lex king)))
 (superclass (m39 (lex person))))

(m42!)
```

```
 CPU time : 0.01

*
; Wizards are persons.
(describe
(assert subclass (build lex "wizard") superclass (build lex "person")
        kn_cat "life"))

(m44! (kn_cat life) (subclass (m43 (lex wizard)))
 (superclass (m39 (lex person))))

(m44!)

 CPU time : 0.01

*
; Barons are persons.
(describe
(assert subclass (build lex "baron") superclass (build lex "person")
        kn_cat "life"))

(m46! (kn_cat life) (subclass (m45 (lex baron)))
 (superclass (m39 (lex person))))

(m46!)

 CPU time : 0.01

*
; Knights are persons.
(describe
(assert subclass (build lex "knight") superclass (build lex "person")
        kn_cat "life"))

(m48! (kn_cat life) (subclass (m47 (lex knight)))
 (superclass (m39 (lex person))))

(m48!)

 CPU time : 0.00

*
; Person is a basic level category
(describe
(assert member (build lex "person")  class (build lex "basic ctgy")
        kn_cat "life"))

(m49! (class (m20 (lex basic ctgy))) (kn_cat life) (member (m39 (lex person))))

(m49!)

 CPU time : 0.01

*
; Something is named 'King Arthur'
(describe
 (assert object #KA proper-name (build lex "King Arthur")
         kn_cat "life"))
```

```
(m51! (kn_cat life) (object b3) (proper-name (m50 (lex King Arthur))))

(m51!)

 CPU time : 0.01

*
; King Arthur is a king
(describe
(assert member *KA class (build lex "king")
        kn_cat "life"))

(m52! (class (m41 (lex king))) (kn_cat life) (member b3))

(m52!)

 CPU time : 0.00

*
; The Round Table is a table
(describe
(assert member (build lex "Round Table") class (build lex "table")
        kn_cat "life"))

(m55! (class (m54 (lex table))) (kn_cat life) (member (m53 (lex Round Table))))

(m55!)

 CPU time : 0.01

*
; King Arthur owns the Round Table
(describe
(assert possessor *KA object (build lex "Round Table") rel (build lex "table")
        kn_cat "life"))

(m56! (kn_cat life) (object (m53 (lex Round Table))) (possessor b3)
 (rel (m54 (lex table))))

(m56!)

 CPU time : 0.01

*
; There is something named 'Excalibur'
(describe
(assert object #Excalibur proper-name (build lex "Excalibur")
        kn_cat "life"))

(m58! (kn_cat life) (object b4) (proper-name (m57 (lex Excalibur))))

(m58!)

 CPU time : 0.00

*
; Excalibur is a sword
```

```
(describe
(assert  member *Excalibur class (build lex "sword")
        kn_cat "life"))

(m60! (class (m59 (lex sword))) (kn_cat life) (member b4))

(m60!)

 CPU time : 0.00

*
; King Arthur owns Excalibur
(describe
(assert object *Excalibur rel (build lex "sword") possessor *KA
        kn_cat "life"))

(m61! (kn_cat life) (object b4) (possessor b3) (rel (m59 (lex sword))))

(m61!)

 CPU time : 0.00

*
; Something is named 'Merlin'
(describe
(assert object #Mer proper-name (build lex "Merlin")
        kn_cat "life"))

(m63! (kn_cat life) (object b5) (proper-name (m62 (lex Merlin))))

(m63!)

 CPU time : 0.01

*
; Merlin is a wizard.
(describe
(assert member *Mer  class (build lex "wizard")
        kn_cat "life"))

(m64! (class (m43 (lex wizard))) (kn_cat life) (member b5))

(m64!)

 CPU time : 0.00

*
; Something is named 'Sir Galahad'
(describe
 (assert object #Galahad proper-name (build lex "Sir Galahad")
         kn_cat "life"))

(m66! (kn_cat life) (object b6) (proper-name (m65 (lex Sir Galahad))))

(m66!)

 CPU time : 0.00
```

```
*
; Sir Galahad is a knight
(describe
(assert member *Galahad  class (build lex "knight")
        kn_cat "life"))

(m67! (class (m47 (lex knight))) (kn_cat life) (member b6))

(m67!)

 CPU time : 0.01

*
; Something is named 'Sir Tristram'
(describe
(assert object #Tris proper-name (build lex "Sir Tristram")
        kn_cat "life"))

(m69! (kn_cat life) (object b7) (proper-name (m68 (lex Sir Tristram))))

(m69!)

 CPU time : 0.01

*
; Sir Tristram is a knight
(describe
(assert member *Tris  class (build lex "knight")
        kn_cat "life"))

(m70! (class (m47 (lex knight))) (kn_cat life) (member b7))

(m70!)

 CPU time : 0.01

*
; Something is named 'Sir Gawain'
(describe
(assert object #SG proper-name (build lex "Sir Gawain")
        kn_cat "life"))

(m72! (kn_cat life) (object b8) (proper-name (m71 (lex Sir Gawain))))

(m72!)

 CPU time : 0.00

*
; Sir Gawain is a knight
(describe
(assert member *SG  class (build lex "knight")
        kn_cat "life"))

(m73! (class (m47 (lex knight))) (kn_cat life) (member b8))

(m73!)
```

```
 CPU time : 0.00

*
; Something is named 'King Ban'
(describe
(assert object #Ban proper-name (build lex "King Ban")
        kn_cat "life"))

(m75! (kn_cat life) (object b9) (proper-name (m74 (lex King Ban))))

(m75!)

 CPU time : 0.01

*
; King Ban is a king
(describe
(assert member *Ban  class (build lex "king")
        kn_cat "life"))

(m76! (class (m41 (lex king))) (kn_cat life) (member b9))

(m76!)

 CPU time : 0.01

*
; Something is named 'King Bors'
(describe
(assert object #Bors proper-name (build lex "King Bors")
        kn_cat "life"))

(m78! (kn_cat life) (object b10) (proper-name (m77 (lex King Bors))))

(m78!)

 CPU time : 0.01

*
; King Bors is a king
(describe
(assert member *Bors  class (build lex "king")
        kn_cat "life"))

(m79! (class (m41 (lex king))) (kn_cat life) (member b10))

(m79!)

 CPU time : 0.00

*
; Something is named 'King Lot'
(describe
(assert object #Lot proper-name (build lex "King Lot")
        kn_cat "life"))

(m81! (kn_cat life) (object b11) (proper-name (m80 (lex King Lot))))
```

```
(m81!)

 CPU time : 0.00

*
; King Lot is a king
(describe
(assert member *Lot  class (build lex "king")
        kn_cat "life"))

(m82! (class (m41 (lex king))) (kn_cat life) (member b11))

(m82!)

 CPU time : 0.00

*
; Sideboards are furniture
(describe
(assert subclass (build lex "sideboard") superclass (build lex "furniture")
        kn_cat "life"))

(m85! (kn_cat life) (subclass (m83 (lex sideboard)))
 (superclass (m84 (lex furniture))))

(m85!)

 CPU time : 0.01

*
; Tables are furniture
(describe
(assert subclass (build lex "table") superclass (build lex "furniture")
        kn_cat "life"))

(m86! (kn_cat life) (subclass (m54 (lex table)))
 (superclass (m84 (lex furniture))))

(m86!)

 CPU time : 0.00

*
; Chairs are furniture
(describe
(assert subclass (build lex "chair") superclass (build lex "furniture")
        kn_cat "life"))

(m88! (kn_cat life) (subclass (m87 (lex chair)))
 (superclass (m84 (lex furniture))))

(m88!)

 CPU time : 0.00

*
; Chair is a basic level category
(describe
```

```
(assert member (build lex "chair")  class (build lex "basic ctgy")
        kn_cat "life"))

(m89! (class (m20 (lex basic ctgy))) (kn_cat life) (member (m87 (lex chair))))

(m89!)

 CPU time : 0.01

*
; Table is a basic level category
(describe
(assert member (build lex "table")  class (build lex "basic ctgy")
        kn_cat "life"))

(m90! (class (m20 (lex basic ctgy))) (kn_cat life) (member (m54 (lex table))))

(m90!)

 CPU time : 0.01

*
; White is a color
(describe
(assert member (build lex "white") class (build lex "color")
        kn_cat "life"))

(m93! (class (m92 (lex color))) (kn_cat life) (member (m91 (lex white))))

(m93!)

 CPU time : 0.00

*
; Black is a color
(describe
(assert member (build lex "black") class (build lex "color")
        kn_cat "life"))

(m95! (class (m92 (lex color))) (kn_cat life) (member (m94 (lex black))))

(m95!)

 CPU time : 0.01

*
; Small is a size
(describe
(assert member (build lex "small") class (build lex "size")
        kn_cat "life"))

(m98! (class (m97 (lex size))) (kn_cat life) (member (m96 (lex small))))

(m98!)

 CPU time : 0.01

*
```

```
; "Small" and "little" are synonyms
(describe
(assert synonym (build lex "small") synonym (build lex "little")
        kn_cat "life"))

(m100! (kn_cat life) (synonym (m99 (lex little)) (m96 (lex small))))

(m100!)

 CPU time : 0.01

*
; Large is a size
(describe
(assert member (build lex "large") class (build lex "size")
        kn_cat "life"))

(m102! (class (m97 (lex size))) (kn_cat life) (member (m101 (lex large))))

(m102!)

 CPU time : 0.01

*
; "Large" and "big" are synonyms
(describe
(assert synonym (build lex "large") synonym (build lex "big")
        kn_cat "life"))

(m104! (kn_cat life) (synonym (m103 (lex big)) (m101 (lex large))))

(m104!)

 CPU time : 0.00

*
; Spears are weapons
(describe
(assert subclass (build lex "spear") superclass (build lex "weapon")
        kn_cat "life"))

(m107! (kn_cat life) (subclass (m105 (lex spear)))
 (superclass (m106 (lex weapon))))

(m107!)

 CPU time : 0.00

*
; Jousts are types of combat
;(describe
;(assert subclass (build lex "joust") superclass (build lex "combat")
;       kn_cat "life"))

; Tourneys are types of combat
(describe
(assert subclass (build lex "tourney") superclass (build lex "combat")
        kn_cat "life"))
```

```
(m110! (kn_cat life) (subclass (m108 (lex tourney)))
 (superclass (m109 (lex combat))))

(m110!)

 CPU time : 0.00

*
; Combat is a type of event
(describe
(assert subclass (build lex "combat") superclass (build lex "event")
        kn_cat "life"))

(m112! (kn_cat life) (subclass (m109 (lex combat)))
 (superclass (m111 (lex event))))

(m112!)

 CPU time : 0.00

*
; Tourney is another word for Tournament
(describe
(assert synonym (build lex "tourney") synonym (build lex "tournament")
        kn_cat "life"))

(m114! (kn_cat life) (synonym (m113 (lex tournament)) (m108 (lex tourney))))

(m114!)

 CPU time : 0.00

*
; "Kill" and "Slay" are synonyms
(describe
(assert synonym (build lex "kill") synonym (build lex "slay") kn_cat "life"))

(m117! (kn_cat life) (synonym (m116 (lex slay)) (m115 (lex kill))))

(m117!)

 CPU time : 0.01

*
;#######################
;;          RULES
;#######################

; Tourneys and Jousts are formal, mock things
(describe
(add forall $event
        ant (;;(build member *event class (build lex "joust"))
             (build member *event class (build lex "tourney")))
        cq  ((build object *event property (build lex "formal"))
             (build object *event property (build lex "mock")))
        kn_cat "life-rule.2"))
```

```
(m120! (forall v1) (ant (p1 (class (m108 (lex tourney))) (member v1)))
 (cq (p3 (object v1) (property (m119 (lex mock))))
  (p2 (object v1) (property (m118 (lex formal)))))
 (kn_cat life-rule.2))

(m120!)

 CPU time : 0.13

*
; If an event is formal then there is some entity that arranges
;  the event.
(describe
(add forall $event
        ant ((build object *event property (build lex "formal"))
             (build member *event class (build lex "event")))
        cq (build agent (build skf "arranger-of" a1 *event)
                  act (build action (build lex "arrange")
                            object *event))
        kn_cat "life-rule.1"))

(m122! (forall v2)
 (ant (p5 (class (m111 (lex event))) (member v2))
  (p4 (object v2) (property (m118 (lex formal)))))
 (cq
  (p8 (act (p7 (action (m121 (lex arrange))) (object v2)))
   (agent (p6 (a1 v2) (skf arranger-of)))))
 (kn_cat life-rule.1))

(m122!)

 CPU time : 0.05

*
; If something is a hound then that thing hunts.
(describe
(add forall $hound1
        ant (build member *hound1 class (build lex "hound"))
        cq (build agent *hound1 act (build action (build lex "hunt")))
        kn_cat "life-rule.1"))

(m125! (forall v3) (ant (p9 (class (m35 (lex hound))) (member v3)))
 (cq (p10 (act (m124 (action (m123 (lex hunt))))) (agent v3)))
 (kn_cat life-rule.1))

(m125!)

 CPU time : 0.02

*
;; If something bays and it is a member of some class then
;;  that class is a subclass of hound
(describe
 (add forall ($bayer $categ)
   &ant ((build agent *bayer act (build action (build lex "bay")))
         (build member *bayer class *categ))
   cq (build subclass *categ superclass (build lex "hound"))))
```

```
(m163! (class (m20 (lex basic ctgy))) (member (m15 (lex deer))))
(m162! (class (m20)) (member (m22 (lex horse))))
(m161! (class (m20)) (member (m29 (lex dog))))
(m160! (class (m12 (lex mammal))) (member b1))
(m159! (class (m10 (lex carnivore))) (member b1))
(m158! (class (m5 (lex vertebrate))) (member b1))
(m157! (class (m4 (lex quadruped))) (member b1))
(m156! (class (m2 (lex phys obj))) (member b1))
(m155! (class (m1 (lex animal))) (member b1))
(m154! (class (m20)) (member (m39 (lex person))))
(m153! (class (m41 (lex king))) (member b3))
(m152! (class (m39)) (member b3))
(m151! (class (m84 (lex furniture))) (member (m53 (lex Round Table))))
(m150! (class (m54 (lex table))) (member (m53)))
(m149! (class (m59 (lex sword))) (member b4))
(m148! (class (m43 (lex wizard))) (member b5))
(m147! (class (m39)) (member b5))
(m146! (class (m47 (lex knight))) (member b6))
(m145! (class (m39)) (member b6))
(m144! (class (m47)) (member b7))
(m143! (class (m39)) (member b7))
(m142! (class (m47)) (member b8))
(m141! (class (m39)) (member b8))
(m140! (class (m41)) (member b9))
(m139! (class (m39)) (member b9))
(m138! (class (m41)) (member b10))
(m137! (class (m39)) (member b10))
(m136! (class (m41)) (member b11))
(m135! (class (m39)) (member b11))
(m134! (class (m20)) (member (m87 (lex chair))))
(m133! (class (m20)) (member (m54)))
(m132! (class (m92 (lex color))) (member (m91 (lex white))))
(m131! (class (m92)) (member (m94 (lex black))))
(m130! (class (m97 (lex size))) (member (m96 (lex small))))
(m129! (class (m97)) (member (m101 (lex large))))
(m128! (forall v5 v4)
 (&ant (p12 (class v5) (member v4))
  (p11 (act (m127 (action (m126 (lex bay))))) (agent v4)))
 (cq (p13 (subclass v5) (superclass (m35 (lex hound))))))
(m40! (class (m39)) (member b2))
(m37! (class (m29)) (member b1))

(m163! m162! m161! m160! m159! m158! m157! m156! m155! m154! m153! m152! m151!
 m150! m149! m148! m147! m146! m145! m144! m143! m142! m141! m140! m139! m138!
 m137! m136! m135! m134! m133! m132! m131! m130! m129! m128! m40! m37!)

 CPU time : 0.17

*
;; Newly inferred information:
;;
;; "Deer" is a basic category
;; "Horse" is a basic category
;; "Dog" is a basic category
;; The dog (b1) is a quadruped
;; The dog (b1) is a mammal
;; The dog (b1) is a vertebrate
;; The dog (b1) is a physical object
```

```
;; The dog (b1) is a animal
;; "Person" is a basic category
;; King Arthur is a king
;; King Arthur is a person
;; The Round Table is a piece of furniture
;; The Round Table is a table
;; Excalibur is a sword
;; Merlin is a wizard
;; Merlin is a person
;; Sir Galahad is a knight
;; Sir Galahad is a person
;; Sir Tristram is a knight
;; Sir Tristram is a person
;; Sir Gawain is a knight
;; Sir Gawain is a person
;; King Ban is a king
;; King Ban is a person
;; King Bors is a king
;; King Bors is a person
;; King Lot is a king
;; King Lot is a person
;; "Chair" is a basic category
;; "Table" is a basic category
;; White is a color
;; Black is a color
;; Small is a size
;; Large is a size

;; If one thing bites another and the biter is a member of some
;;  class then that class is a subclass of animal
(describe
(add forall ($animal1 $bitten *categ)
       &ant ((build agent *animal1 act (build action (build lex "bite")
                                                object *bitten))
             (build member *animal1 class *categ))
        cq (build subclass *categ superclass (build lex "animal"))
        kn_cat "life-rule.1"))

(m165! (forall v7 v6 v5)
 (&ant (p16 (class v5) (member v6))
  (p15 (act (p14 (action (m164 (lex bite))) (object v7))) (agent v6)))
 (cq (p17 (subclass v5) (superclass (m1 (lex animal)))))) (kn_cat life-rule.1))
(m163! (class (m20 (lex basic ctgy))) (member (m15 (lex deer))))
(m162! (class (m20)) (member (m22 (lex horse))))
(m161! (class (m20)) (member (m29 (lex dog))))
(m160! (class (m12 (lex mammal))) (member b1))
(m159! (class (m10 (lex carnivore))) (member b1))
(m158! (class (m5 (lex vertebrate))) (member b1))
(m157! (class (m4 (lex quadruped))) (member b1))
(m156! (class (m2 (lex phys obj))) (member b1))
(m155! (class (m1)) (member b1))
(m154! (class (m20)) (member (m39 (lex person))))
(m153! (class (m41 (lex king))) (member b3))
(m152! (class (m39)) (member b3))
(m151! (class (m84 (lex furniture))) (member (m53 (lex Round Table))))
(m150! (class (m54 (lex table))) (member (m53)))
(m149! (class (m59 (lex sword))) (member b4))
(m148! (class (m43 (lex wizard))) (member b5))
```

```
(m147! (class (m39)) (member b5))
(m146! (class (m47 (lex knight))) (member b6))
(m145! (class (m39)) (member b6))
(m144! (class (m47)) (member b7))
(m143! (class (m39)) (member b7))
(m142! (class (m47)) (member b8))
(m141! (class (m39)) (member b8))
(m140! (class (m41)) (member b9))
(m139! (class (m39)) (member b9))
(m138! (class (m41)) (member b10))
(m137! (class (m39)) (member b10))
(m136! (class (m41)) (member b11))
(m135! (class (m39)) (member b11))
(m134! (class (m20)) (member (m87 (lex chair))))
(m133! (class (m20)) (member (m54)))
(m132! (class (m92 (lex color))) (member (m91 (lex white))))
(m131! (class (m92)) (member (m94 (lex black))))
(m130! (class (m97 (lex size))) (member (m96 (lex small))))
(m129! (class (m97)) (member (m101 (lex large))))
(m40! (class (m39)) (member b2))
(m37! (class (m29)) (member b1))

(m165! m163! m162! m161! m160! m159! m158! m157! m156! m155! m154! m153! m152!
 m151! m150! m149! m148! m147! m146! m145! m144! m143! m142! m141! m140! m139!
 m138! m137! m136! m135! m134! m133! m132! m131! m130! m129! m40! m37!)

 CPU time : 0.33

*
;; Newly inferred information:
;;
;; None

;; If something is an animal and part of another class then
;;   presumably, that class is a subclass of animal
(describe
(add forall (*animal1 $class2)
       &ant ((build member *animal1 class (build lex "animal"))
             (build member *animal1 class *class2))
       cq (build mode   (build lex "presumably")
                 object (build subclass *class2 superclass (build lex "animal")))
       kn_cat "life-rule.1"))

(m189! (mode (m166 (lex presumably)))
 (object
  (m188 (subclass (m10 (lex carnivore))) (superclass (m1 (lex animal))))))
(m187! (mode (m166))
 (object (m186 (subclass (m5 (lex vertebrate))) (superclass (m1)))))
(m185! (mode (m166))
 (object (m184 (subclass (m4 (lex quadruped))) (superclass (m1)))))
(m181! (mode (m166))
 (object (m180 (subclass (m12 (lex mammal))) (superclass (m1)))))
(m179! (mode (m166))
 (object (m178 (subclass (m15 (lex deer))) (superclass (m1)))))
(m177! (mode (m166))
 (object (m176 (subclass (m22 (lex horse))) (superclass (m1)))))
(m173! (mode (m166))
 (object (m172 (subclass (m29 (lex dog))) (superclass (m1)))))
```

```
(m171! (mode (m166))
 (object (m170 (subclass (m35 (lex hound))) (superclass (m1)))))
(m169! (mode (m166))
 (object (m168 (subclass (m2 (lex phys obj))) (superclass (m1)))))
(m167! (forall v8 v6)
 (&ant (p19 (class v8) (member v6)) (p18 (class (m1)) (member v6)))
 (cq (p21 (mode (m166)) (object (p20 (subclass v8) (superclass (m1))))))
 (kn_cat life-rule.1))
(m163! (class (m20 (lex basic ctgy))) (member (m15)))
(m162! (class (m20)) (member (m22)))
(m161! (class (m20)) (member (m29)))
(m160! (class (m12)) (member b1))
(m159! (class (m10)) (member b1))
(m158! (class (m5)) (member b1))
(m157! (class (m4)) (member b1))
(m156! (class (m2)) (member b1))
(m155! (class (m1)) (member b1))
(m154! (class (m20)) (member (m39 (lex person))))
(m153! (class (m41 (lex king))) (member b3))
(m152! (class (m39)) (member b3))
(m151! (class (m84 (lex furniture))) (member (m53 (lex Round Table))))
(m150! (class (m54 (lex table))) (member (m53)))
(m149! (class (m59 (lex sword))) (member b4))
(m148! (class (m43 (lex wizard))) (member b5))
(m147! (class (m39)) (member b5))
(m146! (class (m47 (lex knight))) (member b6))
(m145! (class (m39)) (member b6))
(m144! (class (m47)) (member b7))
(m143! (class (m39)) (member b7))
(m142! (class (m47)) (member b8))
(m141! (class (m39)) (member b8))
(m140! (class (m41)) (member b9))
(m139! (class (m39)) (member b9))
(m138! (class (m41)) (member b10))
(m137! (class (m39)) (member b10))
(m136! (class (m41)) (member b11))
(m135! (class (m39)) (member b11))
(m134! (class (m20)) (member (m87 (lex chair))))
(m133! (class (m20)) (member (m54)))
(m132! (class (m92 (lex color))) (member (m91 (lex white))))
(m131! (class (m92)) (member (m94 (lex black))))
(m130! (class (m97 (lex size))) (member (m96 (lex small))))
(m129! (class (m97)) (member (m101 (lex large))))
(m40! (class (m39)) (member b2))
(m37! (class (m29)) (member b1))

(m189! m187! m185! m181! m179! m177! m173! m171! m169! m167! m163! m162! m161!
 m160! m159! m158! m157! m156! m155! m154! m153! m152! m151! m150! m149! m148!
 m147! m146! m145! m144! m143! m142! m141! m140! m139! m138! m137! m136! m135!
 m134! m133! m132! m131! m130! m129! m40! m37!)

 CPU time : 0.47

*
;; Newly inferred information:
;;
;; Presumably, quadruped is a subclass of animal
;; Presumably, vertebrate is a subclass of animal
```

```
;; Presumably, mammal is a subclass of animal
;; Presumably, deer is a subclass of animal
;; Presumably, hart is a subclass of animal
;; Presumably, horse is a subclass of animal
;; Presumably, dog is a subclass of animal
;; Presumably, hound is a subclass of animal
;; Presumably, physical object is a subclass of animal

;; If something is presumably an animal and is a member of another class then
;;  presumably, that class is a subclass of animal
(describe
(add forall (*animal1 *class2)
        &ant ((build mode (build lex "presumably")
                     object (build member *animal1 class (build lex "animal")))
              (build member *animal1  class *class2))
        cq (build mode (build lex "presumably")
                  object (build subclass *class2
                                 superclass (build lex "animal")))
        kn_cat "life-rule.1"))

(m190! (forall v8 v6)
 (&ant
  (p23 (mode (m166 (lex presumably)))
   (object (p18 (class (m1 (lex animal))) (member v6))))
  (p19 (class v8) (member v6)))
 (cq (p21 (mode (m166)) (object (p20 (subclass v8) (superclass (m1))))))
 (kn_cat life-rule.1))

(m190!)

 CPU time : 0.14

*
;; Newly inferred information:
;;
;; None

;; If something is a mammal and a member of another class then
;; presumably, that class is a subclass of mammal
(describe
(add forall (*animal1 *class2)
        &ant ((build member *animal1  class (build lex "mammal"))
              (build member *animal1  class *class2))
        cq (build mode   (build lex "presumably")
                  object (build subclass *class2 superclass (build lex "mammal")))
        kn_cat "life-rule.1"))

(m213! (mode (m166 (lex presumably)))
 (object (m212 (subclass (m12 (lex mammal))) (superclass (m12)))))
(m211! (mode (m166))
 (object (m210 (subclass (m15 (lex deer))) (superclass (m12)))))
(m209! (mode (m166))
 (object (m208 (subclass (m22 (lex horse))) (superclass (m12)))))
(m207! (mode (m166))
 (object (m206 (subclass (m27 (lex pony))) (superclass (m12)))))
(m205! (mode (m166))
 (object (m204 (subclass (m35 (lex hound))) (superclass (m12)))))
(m203! (mode (m166))
```

```
 (object (m202 (subclass (m1 (lex animal))) (superclass (m12)))))
(m201! (mode (m166))
 (object (m200 (subclass (m4 (lex quadruped))) (superclass (m12)))))
(m199! (mode (m166))
 (object (m198 (subclass (m10 (lex carnivore))) (superclass (m12)))))
(m197! (mode (m166))
 (object (m196 (subclass (m29 (lex dog))) (superclass (m12)))))
(m195! (mode (m166))
 (object (m194 (subclass (m5 (lex vertebrate))) (superclass (m12)))))
(m193! (mode (m166))
 (object (m192 (subclass (m2 (lex phys obj))) (superclass (m12)))))
(m191! (forall v8 v6)
 (&ant (p24 (class (m12)) (member v6)) (p19 (class v8) (member v6)))
 (cq (p26 (mode (m166)) (object (p25 (subclass v8) (superclass (m12))))))
 (kn_cat life-rule.1))
(m160! (class (m12)) (member b1))

(m213! m211! m209! m207! m205! m203! m201! m199! m197! m195! m193! m191! m160!)

 CPU time : 0.38

*
;; Newly inferred information:
;;
;; Presumably, mammal is a subclass of mammal
;; Presumably, deer is a subclass of mammal
;; Presumably, hart is a subclass of mammal
;; Presumably, horse is a subclass of mammal
;; Presumably, pony is a subclass of mammal
;; Presumably, hound is a subclass of mammal
;; Presumably, animal is a subclass of mammal
;; Presumably, vertebrate is a subclass of mammal
;; Presumably, quadruped is a subclass of mammal
;; Presumably, dog is a subclass of mammal
;; Presumably, physical object is a subclass of mammal

;; If something is a mammal, then it presumably bears live young
(describe
(add forall *animal1
        ant (build member *animal1  class (build lex "mammal"))
        cq (build mode (build lex "presumably")
                  object (build agent *animal1 act (build action (build lex "bear")
                                                          object (build lex "live young"))))
        kn_cat "life-rule.1"))

(m219! (mode (m166 (lex presumably)))
 (object
  (m218
   (act (m216 (action (m214 (lex bear))) (object (m215 (lex live young)))))
   (agent b1))))
(m217! (forall v6) (ant (p24 (class (m12 (lex mammal))) (member v6)))
 (cq (p29 (mode (m166)) (object (p28 (act (m216)) (agent v6)))))
 (kn_cat life-rule.1))

(m219! m217!)

 CPU time : 0.05
```

```
*
;; Newly inferred information:
;;
;; The dog (b1) bears live young

;"If something bears something else, the bearer is an animal"
(describe
(add forall (*animal1 $animal2)
        ant (build agent *animal1 act (build action (build lex "bear")
                                            object *animal2))
        cq (build member *animal1  class (build lex "mammal"))
        kn_cat "life-rule.1"))

(m220! (forall v9 v6)
 (ant (p31 (act (p30 (action (m214 (lex bear))) (object v9))) (agent v6)))
 (cq (p24 (class (m12 (lex mammal))) (member v6))) (kn_cat life-rule.1))

(m220!)

 CPU time : 0.05

*
;; Newly inferred information:
;;
;; None

; If there is a person and that person can carry something, then the
;  thing that can be carried has the property "small".
(describe
(add forall ($thingy $person)
       &ant ((build member *person class (build lex "person"))
             (build agent *person act (build action (build lex "carry") object *thingy)))
        cq (build object *thingy property (build lex "small"))
        kn_cat "life-rule.2"))

(m222! (forall v11 v10)
 (&ant (p34 (act (p33 (action (m221 (lex carry))) (object v10))) (agent v11))
  (p32 (class (m39 (lex person))) (member v11)))
 (cq (p35 (object v10) (property (m96 (lex small))))) (kn_cat life-rule.2))
(m152! (class (m39)) (member b3))
(m147! (class (m39)) (member b5))
(m145! (class (m39)) (member b6))
(m143! (class (m39)) (member b7))
(m141! (class (m39)) (member b8))
(m139! (class (m39)) (member b9))
(m137! (class (m39)) (member b10))
(m135! (class (m39)) (member b11))
(m40! (class (m39)) (member b2))

(m222! m152! m147! m145! m143! m141! m139! m137! m135! m40!)

 CPU time : 0.31

*
;; Newly inferred information:
;;
;; None
```

```
; If something wants something then the thing that is wanted is valuable
(describe
(add forall (*thingy *person)
        ant (build agent *person act (build action (build lex "want") object *thingy))
        cq (build object *thingy property (build lex "valuable"))
        kn_cat "life-rule.2"))

(m225! (forall v11 v10)
 (ant (p39 (act (p38 (action (m223 (lex want))) (object v10))) (agent v11)))
 (cq (p40 (object v10) (property (m224 (lex valuable))))) (kn_cat life-rule.2))

(m225!)

 CPU time : 0.04

*
;; Newly inferred information:
;;
;; None

; If something says that it wants another thing, then it actually
;  does want that thing
(describe
(add forall (*thingy *person)
        ant (build agent *person act (build action (build lex "say that")
                                           object (build agent *person
                                                         act (build action (build lex "want")
                                                                   object *thingy))))
        cq (build agent *person
                  act (build action (build lex "want")
                             object *thingy))
        kn_cat "life-rule.2"))

(m227! (forall v11 v10)
 (ant
  (p42
   (act (p41 (action (m226 (lex say that)))
         (object
          (p39 (act (p38 (action (m223 (lex want))) (object v10)))
           (agent v11)))))
   (agent v11)))
 (cq (p39)) (kn_cat life-rule.2))

(m227!)

 CPU time : 0.05

*
;; Newly inferred information:
;;
;; None.

;
(describe
; If a member of some class has a property that is a color,
;  then the class that it is a member of is a subclass of 'physical object'
(add forall ($thing $prop $foo)
        &ant ((build member *foo class *thing)
```

```
            (build object *foo property *prop)
            (build member *prop class (build lex "color")))
        cq (build subclass *thing superclass (build lex "phys obj"))
        kn_cat "intrinsic"))

(m228! (forall v14 v13 v12)
 (&ant (p45 (class (m92 (lex color))) (member v13))
  (p44 (object v14) (property v13)) (p43 (class v12) (member v14)))
 (cq (p46 (subclass v12) (superclass (m2 (lex phys obj))))) (kn_cat intrinsic))
(m163! (class (m20 (lex basic ctgy))) (member (m15 (lex deer))))
(m162! (class (m20)) (member (m22 (lex horse))))
(m161! (class (m20)) (member (m29 (lex dog))))
(m160! (class (m12 (lex mammal))) (member b1))
(m159! (class (m10 (lex carnivore))) (member b1))
(m158! (class (m5 (lex vertebrate))) (member b1))
(m157! (class (m4 (lex quadruped))) (member b1))
(m156! (class (m2)) (member b1))
(m155! (class (m1 (lex animal))) (member b1))
(m154! (class (m20)) (member (m39 (lex person))))
(m153! (class (m41 (lex king))) (member b3))
(m152! (class (m39)) (member b3))
(m151! (class (m84 (lex furniture))) (member (m53 (lex Round Table))))
(m150! (class (m54 (lex table))) (member (m53)))
(m149! (class (m59 (lex sword))) (member b4))
(m148! (class (m43 (lex wizard))) (member b5))
(m147! (class (m39)) (member b5))
(m146! (class (m47 (lex knight))) (member b6))
(m145! (class (m39)) (member b6))
(m144! (class (m47)) (member b7))
(m143! (class (m39)) (member b7))
(m142! (class (m47)) (member b8))
(m141! (class (m39)) (member b8))
(m140! (class (m41)) (member b9))
(m139! (class (m39)) (member b9))
(m138! (class (m41)) (member b10))
(m137! (class (m39)) (member b10))
(m136! (class (m41)) (member b11))
(m135! (class (m39)) (member b11))
(m134! (class (m20)) (member (m87 (lex chair))))
(m133! (class (m20)) (member (m54)))
(m132! (class (m92)) (member (m91 (lex white))))
(m131! (class (m92)) (member (m94 (lex black))))
(m130! (class (m97 (lex size))) (member (m96 (lex small))))
(m129! (class (m97)) (member (m101 (lex large))))
(m40! (class (m39)) (member b2))
(m37! (class (m29)) (member b1))

(m228! m163! m162! m161! m160! m159! m158! m157! m156! m155! m154! m153! m152!
 m151! m150! m149! m148! m147! m146! m145! m144! m143! m142! m141! m140! m139!
 m138! m137! m136! m135! m134! m133! m132! m131! m130! m129! m40! m37!)

 CPU time : 0.55

*
;; Newly inferred information:
;;
;; None.
```

```
; If a member of some class has a property that is a size,
;  then the class that it is a member of is a subclass of 'physical object'
(describe
(add forall (*thing *prop *foo)
        &ant ((build member *foo class *thing)
              (build object *foo property *prop)
              (build member *prop class (build lex "size")))
        cq (build subclass *thing superclass (build lex "phys obj"))
        kn_cat "intrinsic"))

(m229! (forall v14 v13 v12)
 (&ant (p50 (class (m97 (lex size))) (member v13))
  (p44 (object v14) (property v13)) (p43 (class v12) (member v14)))
 (cq (p46 (subclass v12) (superclass (m2 (lex phys obj))))) (kn_cat intrinsic))
(m130! (class (m97)) (member (m96 (lex small))))
(m129! (class (m97)) (member (m101 (lex large))))

(m229! m130! m129!)

 CPU time : 0.10

*
;; Newly inferred information:
;;
;; None.

; A weapon damages
(describe
(add forall $weapon1
        ant (build member *weapon1  class (build lex "weapon"))
        cq (build agent *weapon1 act (build action (build lex "damage")))
        kn_cat "life-rule.1"))

(m232! (forall v15) (ant (p54 (class (m106 (lex weapon))) (member v15)))
 (cq (p55 (act (m231 (action (m230 (lex damage))))) (agent v15)))
 (kn_cat life-rule.1))

(m232!)

 CPU time : 0.08

*
;; Newly inferred information:
;;
;; None.

; If something is an elder then that thing is old and is presumably a person
(describe
(add forall $eld1
        ant (build member *eld1  class (build lex "elder"))
        cq ((build object *eld1 property (build lex "old"))
            (build mode (build lex "presumably")
                   object (build member *eld1 class (build lex "person"))))
        kn_cat "life-rule.1"))

(m235! (forall v16) (ant (p59 (class (m233 (lex elder))) (member v16)))
 (cq
  (p62 (mode (m166 (lex presumably)))
```

```
   (object (p61 (class (m39 (lex person))) (member v16))))
  (p60 (object v16) (property (m234 (lex old)))))
 (kn_cat life-rule.1))

(m235!)

 CPU time : 0.08

*
;; Newly inferred information:
;;
;; None.

; if one thing chases another, the former runs behind the latter
(describe
(add forall ($chaser $chasee)
        ant (build agent *chaser act (build action (build lex "chase") object *chasee))
        cq ((build agent *chaser act (build action (build lex "run")))
            (build object1 *chaser rel (build lex "behind") object2 *chasee))
        kn_cat "life-rule.1"))

(m240! (forall v18 v17)
 (ant (p67 (act (p66 (action (m236 (lex chase))) (object v18))) (agent v17)))
 (cq (p69 (object1 v17) (object2 v18) (rel (m239 (lex behind))))
  (p68 (act (m238 (action (m237 (lex run))))) (agent v17)))
 (kn_cat life-rule.1))

(m240!)

 CPU time : 0.05

*
;; Newly inferred information:
;;
;; None.

End of /projects/stn2/CVA/demos/joust.base demonstration.

 CPU time : 3.51

*

;; So it was ordained, and then a cry was made that every man
;; should try who wished to win the sword.  And for New Year's Day
;; the barons arranged a joust and a tournament, so that all knights
;; who wished to joust or tourney might join. [p. 8]
;;

;;  Someone arranged a something.
(describe
(assert agent #barons1 act (build action (build lex "arrange") object #joust1)
        time #jt1 kn_cat "story"))

(m242! (act (m241 (action (m121 (lex arrange))) (object b13))) (agent b12)
 (kn_cat story) (time b14))

(m242!)
```

```
 CPU time : 0.01

*
;;  The thing happens at some time
(describe
(assert event *joust1 time #jt2 kn_cat "story-comp"))

(m243! (event b13) (kn_cat story-comp) (time b15))

(m243!)

 CPU time : 0.00

*

;;  The time that the event was arranged is before the time that it takes place
(describe
(assert before *jt1 after *jt2 kn_cat "story-comp"))

(m244! (after b15) (before b14) (kn_cat story-comp))

(m244!)

 CPU time : 0.00

*
;;  The arranger is a baron
(describe
(assert member *barons1 class (build lex "baron")
        kn_cat "story"))

(m245! (class (m45 (lex baron))) (kn_cat story) (member b12))

(m245!)

 CPU time : 0.00

*
;;  The event is a joust
(describe
(add member *joust1 class (build lex "joust") kn_cat "story"))

(m248! (class (m246 (lex joust))) (member b13))
(m247! (class (m246)) (kn_cat story) (member b13))

(m248! m247!)

 CPU time : 0.03

*
;; Define "joust" as a noun.
^(
-->defineNoun "joust")
 Definition of joust:
 Actions performed on a joust: baron arrange,
nil

 CPU time : 0.53
```

```
*
;;
;;  If X is a knight and X wishes to joust then X might joust.
;;   (Note: "might" is used in the sense of "is allowed to" here)
(describe
(assert forall $jknight
        &ant ((build member *jknight class (build lex knight))
              (build agent *jknight act
                     (build action (build lex "wish")
                            object (build agent *jnkight act
                                          (build action (build lex "joust"))
                                          time *jt2))))
        cq (build mode (build lex "might")
                  object (build agent *jnkight act (build action (build lex "joust"))
                                time *jt2))
        kn_cat "story"))

(m266! (forall v21)
 (&ant
  (p85
   (act (m263 (action (m260 (lex wish)))
         (object (m262 (act (m261 (action (m246 (lex joust))))) (time b15)))))
   (agent v21))
  (p84 (class (m47 (lex knight))) (member v21)))
 (cq (m265 (mode (m264 (lex might))) (object (m262)))) (kn_cat story))

(m266!)

 CPU time : 0.01

*
;; Define "joust" as a verb
^(
--> defineVerb "joust")
"You want me to define the verb 'joust'.

I'll start by looking at the predicate stucture of the sentences I know that use 'joust'.  Here is

The most common type of sentences I know of that use 'joust' are of the form:
     'A something can joust.'

     'A something can joust something.'

     'A something can joust something to something.'

No superclasses were found for this verb.

 Sorting from the most common predicate case to the least common here is what I know.  I will firs

Now, looking from the bottom up I want to get a sense of the categories that most of the agents, c

"

 CPU time : 0.02

*
;;The barons arranged a something (at the same time that they
```

```
;;  arranged the joust).
(describe
 (assert agent *barons1 act
         (build action (build lex "arrange") object #tourney1)
         time *jt1 kn_cat "story"))

(m268! (act (m267 (action (m121 (lex arrange))) (object b16))) (agent b12)
 (kn_cat story) (time b14))

(m268!)

 CPU time : 0.01

*
;; The event takes place at the same time as the joust.
(describe
(assert event *tourney1 time *jt2 kn_cat "story-comp"))

(m269! (event b16) (kn_cat story-comp) (time b15))

(m269!)

 CPU time : 0.00

*
;; The time that the event was arranged is before the time that
;;   the event took place
(describe
(assert before *jt1 after *jt2 kn_cat "story-comp"))

(m244! (after b15) (before b14) (kn_cat story-comp))

(m244!)

 CPU time : 0.00

*
;; The event is a tourney
(describe
(assert member *tourney1 class (build lex "tourney")
        kn_cat "story"))

(m270! (class (m108 (lex tourney))) (kn_cat story) (member b16))

(m270!)

 CPU time : 0.00

*
;; All knights who wished to tourney might tourney.
;;  forall X if X is a knight and X wishes to tourney then X might tourney.
(describe
(assert forall *jknight
        &ant ((build member *jknight class (build lex knight))
              (build agent *jknight act
                     (build action (build lex "wish")
                            object (build agent *jnkight act
                                          (build action (build lex "tourney")))
```

```
                        time *jt2)))
        cq (build mode (build lex "might")
                  object (build agent *jnkight act
                                (build action (build lex "tourney"))
                                time *jt2))
        kn_cat "story"))

(m276! (forall v21)
 (&ant
  (p86
   (act (m273 (action (m260 (lex wish)))
         (object (m272 (act (m271 (action (m108 (lex tourney)))))))
         (time b15)))
   (agent v21))
  (p84 (class (m47 (lex knight))) (member v21)))
 (cq (m275 (mode (m264 (lex might))) (object (m274 (act (m271)) (time b15)))))
 (kn_cat story))

(m276!)

 CPU time : 0.01

*
;Upon New Year's Day, when the service was done, the barons rode
;onto the field, some to joust and some to tourney.  [p. 8]
;;
;; The barons rode onto a something at a time.
;;
(describe
(assert agent *barons1 act (build action (build lex "ride") onto #field1)
        time #jt3 kn_cat "story"))

(m279! (act (m278 (action (m277 (lex ride))) (onto b17))) (agent b12)
 (kn_cat story) (time b18))

(m279!)

 CPU time : 0.01

*
;;  The thing the barons rode onto is a field
(describe
(assert member *field1 class (build lex "field") kn_cat "story"))

(m281! (class (m280 (lex field))) (kn_cat story) (member b17))

(m281!)

 CPU time : 0.00

*
;; The time the barons rode onto the field is after the time that they
;; arranged the (joust/tourney)
(describe
(assert before *jt1 after *jt3 kn_cat "story-comp"))

(m282! (after b18) (before b14) (kn_cat story-comp))
```

```
(m282!)

 CPU time : 0.01

*
;; The time that the barons rode onto the field is after the time that
;;   the (joust/tourney) takes place
(describe
(assert before *jt3 after *jt2 kn_cat "story-comp"))

(m283! (after b15) (before b18) (kn_cat story-comp))

(m283!)

 CPU time : 0.00

*
;; Some barons rode to joust on the field.
(describe
 (assert object1 (assert agent #barons2 act
                         (build action (build lex "ride") onto *field1)
                         time *jt3 kn_cat "story")
   rel (build lex "purpose")
   object2 (build agent *barons2 act (build action (build lex "joust")
                                                   place *field1)
                  time *jt2)
   kn_cat "story"))

(m288! (kn_cat story)
 (object1
  (m284! (act (m278 (action (m277 (lex ride))) (onto b17))) (agent b19)
   (kn_cat story) (time b18)))
 (object2
  (m287 (act (m286 (action (m246 (lex joust))) (place b17))) (agent b19)
   (time b15)))
 (rel (m285 (lex purpose))))

(m288!)

 CPU time : 0.02

*
;; The barons who rode to joust are in the set of barons who arranged the joust
(describe
(add    subset *barons2 superset *barons1 kn_cat "story"))

(m289! (kn_cat story) (subset b19) (superset b12))

(m289!)

 CPU time : 0.03

*
;; The second set of barons joust
(describe
 (assert agent *barons2 act (build action (build lex "joust"))
         time *jt2 kn_cat "story-comp"))
```

```
(m290! (act (m261 (action (m246 (lex joust))))) (agent b19)
 (kn_cat story-comp) (time b15))

(m290!)

 CPU time : 0.00

*
;; Define "joust" as a verb
^(
-->defineVerb "joust")
"You want me to define the verb 'joust'.

I'll start by looking at the predicate stucture of the sentences I know that use 'joust'.  Here is

The most common type of sentences I know of that use 'joust' are of the form:
     'A something can joust.'

No superclasses were found for this verb.

 Sorting from the most common predicate case to the least common here is what I know.  I will firs

Now, looking from the bottom up I want to get a sense of the categories that most of the agents, o

"

 CPU time : 0.17

*
;; Others of the barons rode to tourney on the field.
(describe
 (assert object1 (assert agent #barons3 act
                         (build action (build lex "ride")
                                onto *field1)
                         time *jt3 kn_cat "story")
   rel (build lex "purpose")
   object2 (build agent *barons3 act
                  (build action (build lex "tourney")
                         place *field1)
                  time *jt2)
   kn_cat "story"))

(m298! (kn_cat story)
 (object1
  (m295! (act (m278 (action (m277 (lex ride))) (onto b17))) (agent b20)
   (kn_cat story) (time b18)))
 (object2
  (m297 (act (m296 (action (m108 (lex tourney))) (place b17))) (agent b20)
   (time b15)))
 (rel (m285 (lex purpose))))

(m298!)

 CPU time : 0.01

*
;; The third group of barons is a subset of the group of barons who arranged
;;   the joust
```

```
(describe
(assert subset *barons3 superset *barons1 kn_cat "story"))

(m299! (kn_cat story) (subset b20) (superset b12))

(m299!)

 CPU time : 0.01

*
;; The third group of barons tourney.
(describe
 (assert agent *barons3 act (build action (build lex "tourney"))
         time *jt2 kn_cat "story-comp"))

(m300! (act (m271 (action (m108 (lex tourney))))) (agent b20)
 (kn_cat story-comp) (time b15))

(m300!)

 CPU time : 0.01

*
;And it happened that Sir Ector ... rode unto the joust;
;; with him rode his son Sir Kay and young Arthur.  [p. 8]
;;
;;  Something rode to the joust.
(describe
 (assert agent #Ector act
         (build action (build lex "ride") to *joust1)
         time *jt3 kn_cat "story"))

(m302! (act (m301 (action (m277 (lex ride))) (to b13))) (agent b21)
 (kn_cat story) (time b18))

(m302!)

 CPU time : 0.00

*
;; The thing that rode into the joust is named "Sir Ector"
(describe
(assert object *Ector proper-name (build lex "Sir Ector") kn_cat "story"))

(m304! (kn_cat story) (object b21) (proper-name (m303 (lex Sir Ector))))

(m304!)

 CPU time : 0.00

*
;; Sir Ector is a knight
(describe
(assert member *Ector class (build lex "knight")
        kn_cat "story-comp"))

(m305! (class (m47 (lex knight))) (kn_cat story-comp) (member b21))
```

```
(m305!)

 CPU time : 0.01

*
;; Something rode to the joust.
(describe
 (assert agent #Kay act
         (build action (build lex "ride") to *joust1)
         time *jt3 kn_cat "story"))

(m306! (act (m301 (action (m277 (lex ride))) (to b13))) (agent b22)
 (kn_cat story) (time b18))

(m306!)

 CPU time : 0.01

*
;; The thing that rode to the joust is named "Sir Kay"
(describe
(assert object *Kay proper-name (build lex "Sir Kay") kn_cat "story"))

(m308! (kn_cat story) (object b22) (proper-name (m307 (lex Sir Kay))))

(m308!)

 CPU time : 0.00

*
;; Sir Kay is a knight
(describe
(assert member *Kay class (build lex "knight")
        kn_cat "story-comp"))

(m309! (class (m47 (lex knight))) (kn_cat story-comp) (member b22))

(m309!)

 CPU time : 0.00

*
;; Sir Kay is the son of Sir Hector.
(describe
(assert object1 *Kay rel (build lex "son") object2 *Ector kn_cat "story"))

(m311! (kn_cat story) (object1 b22) (object2 b21) (rel (m310 (lex son))))

(m311!)

 CPU time : 0.01

*
;; Sir Ector is the father of Sir Kay
(describe
(assert object1 *Ector rel (build lex "father") object2 *Kay
        kn_cat "story-comp"))
```

```
(m313! (kn_cat story-comp) (object1 b21) (object2 b22)
 (rel (m312 (lex father))))

(m313!)

 CPU time : 0.00

*
;; Someone rode to the joust.
(describe
(assert agent #Arthur act (build action (build lex "ride") to *joust1)
        time *jt3 kn_cat "story"))

(m314! (act (m301 (action (m277 (lex ride))) (to b13))) (agent b23)
 (kn_cat story) (time b18))

(m314!)

 CPU time : 0.00

*
;; The thing that rode to the joust is named "Arthur"
(describe
(assert object *Arthur proper-name (build lex "Arthur") kn_cat "story"))

(m316! (kn_cat story) (object b23) (proper-name (m315 (lex Arthur))))

(m316!)

 CPU time : 0.01

*
;; Arthur is young
(describe
(assert object *Arthur property (build lex "young") kn_cat "story"))

(m318! (kn_cat story) (object b23) (property (m317 (lex young))))

(m318!)

 CPU time : 0.01

*
;; Arthur is a person
(describe
(assert member *Arthur class (build lex "person") kn_cat "story-comp"))

(m319! (class (m39 (lex person))) (kn_cat story-comp) (member b23))

(m319!)

 CPU time : 0.01

*
;; Define "joust" as a noun.
^(
--> defineNoun "joust")
 Definition of joust:
```

```
 Actions performed on a joust: baron arrange,
nil

 CPU time : 0.28

*
;; But as they rode toward the joust Sir Kay lacked his sword, for
;; he had left it at his father's lodging ... [p. 8]
;;
;; Sir Kay lacked his something because he had left it somewhere.
(describe
 (assert
     effect (assert agent *Kay act
                    (build action (build lex "lack") object #kaysword)
                    time *jt3 kn_cat "story")
     cause (assert agent *Kay act
                   (build action (build lex "leave") object *kaysword
                          place #lodge1)
                   time #jt4 kn_cat "story")
        kn_cat "story"))

(m328!
 (cause
  (m327! (act (m326 (action (m325 (lex leave))) (object b24) (place b25)))
   (agent b22) (kn_cat story) (time b26)))
 (effect
  (m324! (act (m323 (action (m322 (lex lack))) (object b24))) (agent b22)
   (kn_cat story) (time b18)))
 (kn_cat story))

(m328!)

 CPU time : 0.02

*
;; The thing that Sir Kay left somewhere is his sword.
(describe
 (assert object *kaysword rel (build lex "sword")
         possessor *Kay kn_cat "story"))

(m329! (kn_cat story) (object b24) (possessor b22) (rel (m59 (lex sword))))

(m329!)

 CPU time : 0.00

*
;; Sir Kay's sword is a sword.
(describe
(assert member *kaysword class (build lex "sword") kn_cat "story"))

(m330! (class (m59 (lex sword))) (kn_cat story) (member b24))

(m330!)

 CPU time : 0.01

*
```

```
;; The place that Sir Kay left his sword is Sir Ector's lodging
(describe
(assert object *lodge1 rel (build lex "lodging") possessor *Ector
        kn_cat "story"))

(m332! (kn_cat story) (object b25) (possessor b21) (rel (m331 (lex lodging))))

(m332!)

 CPU time : 0.00

*
;; Sir Ector's lodging is a lodging
(describe
(assert member *lodge1 class (build lex "lodging") kn_cat "story"))

(m333! (class (m331 (lex lodging))) (kn_cat story) (member b25))

(m333!)

 CPU time : 0.01

*
;; The time that Sir Kay left his sword is before the time that
;;  he came to the field
(describe
(assert before *jt4 after *jt3 kn_cat "story-comp"))

(m334! (after b18) (before b26) (kn_cat story-comp))

(m334!)

 CPU time : 0.01

*
;; But when [Arthur] reached home the lady and all the others were
;; out to see the jousting, and he could not enter.  [p. 8]
;;
;; Arthur arrived at the lodging.
(describe
 (assert agent *Arthur act
         (build action (build lex "arrive") place *lodge1)
         time #jt5
         kn_cat "story"))

(m337! (act (m336 (action (m335 (lex arrive))) (place b25))) (agent b23)
 (kn_cat story) (time b27))

(m337!)

 CPU time : 0.02

*
;; Arthur arrived at the lodging after he rode onto the field
(describe
(assert before *jt3 after *jt5 kn_cat "story-comp"))

(m338! (after b27) (before b18) (kn_cat story-comp))
```

```
(m338!)

 CPU time : 0.00

*
;; Arthur could not enter the lodging, because no-one was in the lodging.
(describe
(assert cause (assert forall $human
                      ant (build member *human class (build lex "person"))
                      cq (build min 0 max 0
                                arg (build object1 *human
                                           rel (build lex "in")
                                           object2 *lodge1
                                           time *jt5))
                     kn_cat "story") = stage1
        effect (assert min 0 max 0
                       arg (build mode (build lex "can")
                                  object (build agent *Arthur
                                                act (build action
                                                           (build lex "enter")
                                                           object *lodge1)
                                                time *jt5))
                       kn_cat "story")
        kn_cat "story"))

(m347!
 (cause
  (m340! (forall v24) (ant (p91 (class (m39 (lex person))) (member v24)))
   (cq
    (p93 (min 0) (max 0)
     (arg
      (p92 (object1 v24) (object2 b25) (rel (m339 (lex in))) (time b27)))))
   (kn_cat story)))
 (effect
  (m346! (min 0) (max 0)
   (arg
    (m345 (mode (m341 (lex can)))
     (object
      (m344 (act (m343 (action (m342 (lex enter))) (object b25))) (agent b23)
       (time b27)))))
   (kn_cat story)))
 (kn_cat story))

(m347!)

 CPU time : 0.03

*
;; No-one was in the lodging, because everyone had left.
(describe
(assert effect *stage1
        cause (assert forall (*human $timev)
                      &ant ((build member *human class (build lex "person"))
                            (build object1 *human rel (build lex "in")
                                   object2 *lodge1 time *timev)
                            (build before *timev after *jt5))
                      cq ((build agent *human act
```

```
                                (build action (build lex "leave")
                                       place *lodge1)
                                time (build skf "leave-time"
                                            a1 *human a2 *lodge1) = gotime1)
                           (build before *timev after *gotime1)
                           (build before *gotime1 after *jt5))
                     kn_cat "story") = stage2
        kn_cat "story"))

(m350!
 (cause
  (m349! (forall v25 v24)
   (&ant (p95 (after b27) (before v25))
    (p94 (object1 v24) (object2 b25) (rel (m339 (lex in))) (time v25))
    (p91 (class (m39 (lex person))) (member v24)))
   (cq (p99 (after b27) (before (p96 (a1 v24) (a2 b25) (skf leave-time))))
    (p98 (after (p96)) (before v25))
    (p97 (act (m348 (action (m325 (lex leave))) (place b25))) (agent v24)
     (time (p96))))
   (kn_cat story)))
 (effect
  (m340! (forall v24) (ant (p91))
   (cq
    (p93 (min 0) (max 0)
     (arg (p92 (object1 v24) (object2 b25) (rel (m339)) (time b27)))))
   (kn_cat story)))
 (kn_cat story))

(m350!)

 CPU time : 0.02

*
;; Everyone was out, because they wished to see the joust.
(describe
 (assert forall *human
        &ant ((build member *human class (build lex "person"))
              (build object1 *human rel (build lex "in") object2 *lodge1))
         cq (build agent *human act
                   (build action (build lex "wish")
                          object (build agent *human act
                                        (build action (build lex "see")
                                               object *joust1))))
        kn_cat "story") = stage3)

(m353! (forall v24)
 (&ant (p100 (object1 v24) (object2 b25) (rel (m339 (lex in))))
  (p91 (class (m39 (lex person))) (member v24)))
 (cq
  (p103
   (act (p102 (action (m260 (lex wish)))
         (object
          (p101 (act (m352 (action (m351 (lex see))) (object b13)))
           (agent v24)))))
   (agent v24)))
 (kn_cat story))

(m353!)
```

```
 CPU time : 0.02

*
;; Since everyone was out because they wished to see the joust,
;;  no one was in the lodging (because they all had left)
(describe
(assert effect *stage2
        cause *stage3
        kn_cat "story"))

(m354!
 (cause
  (m353! (forall v24)
   (&ant (p100 (object1 v24) (object2 b25) (rel (m339 (lex in))))
    (p91 (class (m39 (lex person))) (member v24)))
   (cq
    (p103
     (act (p102 (action (m260 (lex wish)))
           (object
            (p101 (act (m352 (action (m351 (lex see))) (object b13)))
             (agent v24)))))
     (agent v24)))
   (kn_cat story)))
 (effect
  (m349! (forall v25 v24)
   (&ant (p95 (after b27) (before v25))
    (p94 (object1 v24) (object2 b25) (rel (m339)) (time v25)) (p91))
   (cq (p99 (after b27) (before (p96 (a1 v24) (a2 b25) (skf leave-time))))
    (p98 (after (p96)) (before v25))
    (p97 (act (m348 (action (m325 (lex leave))) (place b25))) (agent v24)
     (time (p96))))
   (kn_cat story)))
 (kn_cat story))

(m354!)

 CPU time : 0.02

*
;; Define "joust" as a noun
 ^(
-->defineNoun "joust")
 Definition of joust:
 Actions performed on a joust: baron arrange,
nil

 CPU time : 1.09

*
;;  Then the king arranged for a great feast and let cry a great
;;  joust. ... as soon as they had washed and risen, all
;;  knights who would joust made them ready; when they were ready on
;;  horseback, there were seven hundred knights.  [p. 16]
;;
;; Something announced a something.
(describe
 (assert agent #kingj1 act
```

```
        (build action (build lex "announce") object #joust2)
        time #jt6 kn_cat "story"))

(m373! (act (m372 (action (m371 (lex announce))) (object b29))) (agent b28)
 (kn_cat story) (time b30))

(m373!)

 CPU time : 0.01

*
;; The thing that announced is a king
(describe
(assert member *kingj1 class (build lex "king") kn_cat "story"))

(m374! (class (m41 (lex king))) (kn_cat story) (member b28))

(m374!)

 CPU time : 0.01

*
;; The thing that the king announced is a joust
(describe
(assert member *joust2 class (build lex "joust") kn_cat "story"))

(m375! (class (m246 (lex joust))) (kn_cat story) (member b29))

(m375!)

 CPU time : 0.01

*
;; The joust is great
(describe
(assert object *joust2 property (build lex "great") kn_cat "story"))

(m377! (kn_cat story) (object b29) (property (m376 (lex great))))

(m377!)

 CPU time : 0.00

*
;; Define "joust" as a noun
 ^(
-->defineNoun "joust")
 Definition of joust:
 Actions performed on a joust: king announce, baron arrange,
 Possible Properties: great,
nil

 CPU time : 0.94

*
;; All the knights who wished to joust prepared to joust.
(describe
(assert forall *jknight
```

```
        &ant ((build member *jknight class (build lex knight))
              (build agent *jknight act
                     (build action (build lex "wish")
                            object (build agent *jnkight act
                                          (build action (build lex "joust"))))
                     time *jt6))
        cq (build agent *jknght act
                  (build action (build lex "prepare")
                         object (agent *jknight act
                                       (build action (build lex "joust")))))
        kn_cat "story"))

(m399! (forall v21)
 (&ant
  (p122
   (act (m397 (action (m260 (lex wish)))
         (object (m396 (act (m261 (action (m246 (lex joust))))))))
   (agent v21) (time b30))
  (p84 (class (m47 (lex knight))) (member v21)))
 (cq
  (p124
   (act (p123 (action (m398 (lex prepare))) (object act agent (m261) v21)))))
 (kn_cat story))

(m399!)

 CPU time : 0.01

*
;; Define "joust" as a verb
 ^(
--> defineVerb "joust")
"You want me to define the verb 'joust'.

I'll start by looking at the predicate stucture of the sentences I know that use 'joust'.  Here is

The most common type of sentences I know of that use 'joust' are of the form:
     'A something can joust.'

No superclasses were found for this verb.

 Sorting from the most common predicate case to the least common here is what I know.  I will firs

Now, looking from the bottom up I want to get a sense of the categories that most of the agents, o

"

 CPU time : 0.03

*
;; With that the knight came out of the pavilion and said, 'Fair knight,
;; why smote ye down my shield?'
;; 'Because I will joust with you,' said Gryflette.
;; 'It is better that ye do not,' said the knight, 'for ye are young and
;; lately made knight, and your might is nothing to mine.'
;; 'As for that," said Gryflette, 'I will joust with you.'  [p. 33]
;;
;; A something came from something else.
```

```
(describe
 (assert agent #blknt1 act
         (build action (build lex "come") from #jpavilion1)
         time #jt7 kn_cat "story"))

(m402! (act (m401 (action (m400 (lex come))) (from b32))) (agent b31)
 (kn_cat story) (time b33))

(m402!)

 CPU time : 0.01

*
;; The thing that came is a knight
(describe
(assert member *blknt1 class (build lex "knight") kn_cat "story"))

(m403! (class (m47 (lex knight))) (kn_cat story) (member b31))

(m403!)

 CPU time : 0.01

*
;; The thing the knight came from is a pavilion
(describe
(assert member *jpavilion1 class (build lex "pavilion")
        kn_cat "story"))

(m405! (class (m404 (lex pavilion))) (kn_cat story) (member b32))

(m405!)

 CPU time : 0.00

*
;; The knight asked something why he smote down the something else.
(describe
 (assert agent *blknt1 act
         (build action (build lex "ask") indobj #Gryf
                object (build rel (build lex "purpose")
                              object2 (assert agent *Gryf act
                                              (build action (build lex "smite")
                                                     object #knt1shield
                                                     direction
                                                       (build lex "down"))
                                              time #jt8 kn_cat "story") = oddact1
                              object1 (build skf "reason-for" a1 *oddact1)))
        time *jt7 kn_cat "story"))

(m414!
 (act (m413 (action (m406 (lex ask))) (indobj b34)
       (object
        (m412
         (object1
          (m411
           (a1
            (m410!
```

```
            (act (m409 (action (m407 (lex smite)))
                  (direction (m408 (lex down))) (object b35)))
            (agent b34) (kn_cat story) (time b36)))
          (skf reason-for)))
        (object2 (m410!)) (rel (m285 (lex purpose)))))))
 (agent b31) (kn_cat story) (time b33))

(m414!)

 CPU time : 0.02

*
;; The thing that the knight asked is named "Sir Gryflette"
(describe
(assert object *Gryf proper-name (build lex "Sir Gryflette") kn_cat "story"))

(m416! (kn_cat story) (object b34) (proper-name (m415 (lex Sir Gryflette))))

(m416!)

 CPU time : 0.01

*
;; The thing that Sir Gryflette smote down is a shield
(describe
(assert object *knt1shield rel (build lex "shield") possessor *blknt1
        kn_cat "story"))

(m418! (kn_cat story) (object b35) (possessor b31) (rel (m417 (lex shield))))

(m418!)

 CPU time : 0.00

*
;; The shield that Sir Gryflette smote down is a shield
(describe
(assert member *knt1shield class (build lex "shield") kn_cat "story"))

(m419! (class (m417 (lex shield))) (kn_cat story) (member b35))

(m419!)

 CPU time : 0.00

*
;; Sir Gryflette is a knight
(describe
(assert member *Gryf class (build lex "knight")
        kn_cat "story-comp"))

(m420! (class (m47 (lex knight))) (kn_cat story-comp) (member b34))

(m420!)

 CPU time : 0.01

*
```

```
;; The knight asked Sir Gryflette about the sheild after he came from
;;  the pavilion
(describe
(assert before *jt8 after *jt7 kn_cat "story-comp"))

(m421! (after b33) (before b36) (kn_cat story-comp))

(m421!)

 CPU time : 0.00

*
;; Sir Gryflette said that he would joust with the knight.
(describe
 (assert agent *Gryf act
        (build action (build lex "say that")
               object (build agent *Gryf act
                             (build action (build lex "joust") against *blknt1)
                      time #jt9))
        time #jt10 kn_cat "story"))

(m425!
 (act (m424 (action (m226 (lex say that)))
       (object
        (m423 (act (m422 (action (m246 (lex joust))) (against b31)))
         (agent b34) (time b37)))))
 (agent b34) (kn_cat story) (time b38))

(m425!)

 CPU time : 0.01

*
;; Sir Gryflette said that he would joust with the knight before
;;  the joust actually took place
(describe (assert before *jt10 after *jt9 kn_cat "story"))

(m426! (after b37) (before b38) (kn_cat story))

(m426!)

 CPU time : 0.00

*
;; The joust took place after the knight came from the pavilion
(describe (assert before *jt8 after *jt10 kn_cat "story-comp"))

(m427! (after b38) (before b36) (kn_cat story-comp))

(m427!)

 CPU time : 0.00

*
;; The knight said that Sir Gryflette should not joust with him,
;; because Sir Gryflette was young.
(describe
 (assert agent *blknt1 act
```

```
        (build action (build lex "say that")
               object (build cause (build object *Gryf property (build lex "young"))
                             effect (build mode (build lex "should")
                                           object (build min 0 max 0
                                                         arg (build agent *Gryf
                                                                    act (build action (build lex
                                                                                            against *blknt1)
                                                                    time *jt9)))))
               time #jt11 kn_cat "story"))

(m434!
 (act (m433 (action (m226 (lex say that)))
       (object
        (m432 (cause (m428 (object b34) (property (m317 (lex young)))))
         (effect
          (m431 (mode (m429 (lex should)))
           (object
            (m430 (min 0) (max 0)
             (arg
              (m423 (act (m422 (action (m246 (lex joust))) (against b31)))
               (agent b34) (time b37)))))))))))
 (agent b31) (kn_cat story) (time b39))

(m434!)

 CPU time : 0.02

*
;; The knight says that Sir Gryflette should not joust before the joust
;;  takes place.
(describe (assert before *jt11 after *jt9 kn_cat "story"))

(m435! (after b37) (before b39) (kn_cat story))

(m435!)

 CPU time : 0.00

*
;; The knight says that Sir Gryflette should not joust before
;;  Sir Gryflette offers to joust
(describe (assert before *jt10 after *jt11 kn_cat "story-comp"))

(m436! (after b39) (before b38) (kn_cat story-comp))

(m436!)

 CPU time : 0.00

*
;; The knight said that Sir Gryflette should not joust with him,
;; because something was greater than something else.
(describe
 (assert agent *blknt1 act
         (build action (build lex "say that")
                object (build cause (build object1 #knt1might rel (build lex "greater")
                                           object2 #Gryfmight)
                              effect (build mode (build lex "should")
```

```
                                              object (build min 0 max 0
                                                       arg (build agent *Gryf
                                                                  act (build action
                                                                           (build lex "joust"
                                                                           against *blknt1)
                                                                  time *jt9)))))
        time *jt11 kn_cat "story"))

(m441!
 (act (m440 (action (m226 (lex say that)))
       (object
        (m439
         (cause (m438 (object1 b40) (object2 b41) (rel (m437 (lex greater)))))
         (effect
          (m431 (mode (m429 (lex should)))
           (object
            (m430 (min 0) (max 0)
             (arg
              (m423 (act (m422 (action (m246 (lex joust))) (against b31)))
               (agent b34) (time b37)))))))))
 (agent b31) (kn_cat story) (time b39))

(m441!)

 CPU time : 0.01

*
;; The greater thing is the knight's might
(describe (assert object *knt1might rel (build lex "might") possessor *blknt1
                  kn_cat "story"))

(m442! (kn_cat story) (object b40) (possessor b31) (rel (m264 (lex might))))

(m442!)

 CPU time : 0.00

*
;; The lesser thing is Sir Gryflette's might
(describe (assert object *Gryfmight rel (build lex "might") possessor *Gryf
                  kn_cat "story"))

(m443! (kn_cat story) (object b41) (possessor b34) (rel (m264 (lex might))))

(m443!)

 CPU time : 0.03

*
;; Sir Gryflette said that he would joust with the knight.
(describe
 (assert agent *Gryf act
         (build action (build lex "say that")
                object (build agent *Gryf act
                              (build action (build lex "joust")
                                     against *blknt1)
                       time *jt9))
        time #jt12 kn_cat "story"))
```

```
(m444!
 (act (m424 (action (m226 (lex say that)))
       (object
        (m423 (act (m422 (action (m246 (lex joust))) (against b31)))
         (agent b34) (time b37)))))
 (agent b34) (kn_cat story) (time b42))

(m444!)

 CPU time : 0.00

*
;; Sir Gryflette said that he would joust before the joust actually
;;  took place
(describe (assert before *jt12 after *jt9 kn_cat "story"))

(m445! (after b37) (before b42) (kn_cat story))

(m445!)

 CPU time : 0.00

*
;; Sir Gryflette said that he would joust after the knight says that
;;  they should not joust
(describe (assert before *jt11 after *jt12 kn_cat "story-comp"))

(m446! (after b42) (before b39) (kn_cat story-comp))

(m446!)

 CPU time : 0.01

*
;; Sir Gryflette jousts against the knight
(describe
 (assert agent *Gryf act
         (build action (build lex "joust") against *blknt1)
         time *jt9 kn_cat "story-comp"))

(m447! (act (m422 (action (m246 (lex joust))) (against b31))) (agent b34)
 (kn_cat story-comp) (time b37))

(m447!)

 CPU time : 0.01

*
;; The knight jousts agains Sir Gryflette
(describe
 (assert agent *blknt1 act
         (build action (build lex "joust") against *Gryf)
         time *jt9 kn_cat "story-comp"))

(m449! (act (m448 (action (m246 (lex joust))) (against b34))) (agent b31)
 (kn_cat story-comp) (time b37))
```

```
(m449!)

 CPU time : 0.00

*
;; Define "joust" as a verb
 ^(
-->defineVerb "joust")
"You want me to define the verb 'joust'.

I'll start by looking at the predicate stucture of the sentences I know that use 'joust'.  Here is

The most common type of sentences I know of that use 'joust' are of the form:
     'A something can joust.'

No superclasses were found for this verb.

 Sorting from the most common predicate case to the least common here is what I know.  I will firs

A basic ctgy can joust.

Now, looking from the bottom up I want to get a sense of the categories that most of the agents, o

A knight can joust.

"

 CPU time : 0.56

*
;; As they were thus talking, they came to the fountain and the
;; rich pavilion there by it.  Then King Arthur was aware that a
;; knight sat armed in a chair.
;; 'Sir Knight,' said Arthur, 'for what cause abidest thou here, so
;; that no knight may ride this way unless he joust with thee?  I
;; advise thee to leave that custom.'\fP  [They ride at each other
;; with spears.]  \fBThen Arthur set his hand upon his sword.
;; 'Nay,' said the knight, 'ye shall do better. Ye are as passing
;; good a jouster as I ever met with, and for the love of the High
;; Order of Knighthood let us joust again.'  [p. 34]
;;
;;  Something sat in something.
(describe
(assert agent #blknt2 act (build action (build lex "sit") in #knt2chair)
        time #jt13 kn_cat "story"))

(m458! (act (m457 (action (m456 (lex sit))) (in b44))) (agent b43)
 (kn_cat story) (time b45))

(m458!)

 CPU time : 0.01

*
;; The thing that sat is a knight
(describe
(assert member *blknt2 class (build lex "knight")
        kn_cat "story"))
```

```
(m459! (class (m47 (lex knight))) (kn_cat story) (member b43))

(m459!)

 CPU time : 0.00

*
;; The thing that the knight sat in is a chair
(describe
(assert member *knt2chair class (build lex "chair") kn_cat "story"))

(m460! (class (m87 (lex chair))) (kn_cat story) (member b44))

(m460!)

 CPU time : 0.00

*
;; The knight is armed
(describe
(assert object *blknt2 property (build lex "armed")
        time *jt13 kn_cat "story"))

(m462! (kn_cat story) (object b43) (property (m461 (lex armed))) (time b45))

(m462!)

 CPU time : 0.01

*
;; King-Arthur asked the knight why he sat there.
(describe
 (assert agent *KA act
         (build action (build lex "ask") indobj *blknt2
                object (build rel (build lex "purpose")
                              object2 (build agent *blknt2 act
                                             (build action (build lex "sit")))
                              object1 (build skf "reason-for"
                                             a1 (build agent *blknt2 act
                                                        (build action
                                                               (build lex "sit")
                                                               )))))
         time *jt13 kn_cat "story"))

(m468!
 (act (m467 (action (m406 (lex ask))) (indobj b43)
       (object
        (m466
         (object1
          (m465 (a1 (m464 (act (m463 (action (m456 (lex sit))))) (agent b43)))
           (skf reason-for)))
         (object2 (m464)) (rel (m285 (lex purpose)))))))
 (agent b3) (kn_cat story) (time b45))

(m468!)

 CPU time : 0.01
```

```
*
;; King-Arthur asked the knight why he required all knights to joust with him
;; if they wished to pass.
(describe
 (assert agent *KA act
         (build action (build lex "ask") indobj *blknt2
                object (build rel (build lex "purpose")
                              object2
                               (build agent *blknt2 act
                                      (build action (build lex "require")
                                             object
                                              (build forall *jknight
                                                     ant (build agent *jknight
                                                               act (build
                                                                    action (build lex "wish")
                                                                    object (build agent *jknight
                                                                                  act (build act
                                                                                           obj

                                                     cq (build agent *jknight
                                                               act (build action (build lex "joust
                                                               against *blknt2))) = oddact2
                              object1 (build skf "reason-for" a1 *oddact2))))
         time *jt13 kn_cat "story"))

(m478!
 (act (m477 (action (m406 (lex ask))) (indobj b43)
       (object
        (m476
         (object2
          (m475
           (act (m473 (action (m469 (lex require)))
                 (object
                  (m472 (forall v21)
                   (ant
                    (p127
                     (act (p126 (action (m260 (lex wish)))
                           (object
                            (p125
                             (act (m471 (action (m470 (lex pass)))
                                   (object b43)))
                             (agent v21)))))
                     (agent v21)))
                   (cq
                    (p128 (act (m261 (action (m246 (lex joust)))))
                     (against b43) (agent v21)))))))
            (agent b43) (object1 (m474 (a1 (m473)) (skf reason-for)))))
         (rel (m285 (lex purpose)))))))
 (agent b3) (kn_cat story) (time b45))

(m478!)

 CPU time : 0.02

*
;;
;; King-Arthur said that the knight should not require all knights to joust with him.
(describe
```

```
 (assert agent *KA act
         (build action (build lex "say that")
                object (build mode (build lex "should")
                              object (build min 0 max 0
                                            arg (build agent *blknt2 act
                                                       (build action (build lex "require")
                                                              object (build forall *jknight
                                                                            ant (build agent *jkni
                                                                                       (build acti
                                                                                              obje


                                                                            cq (build agent *jknig
                                                                                       (build actio
                                                                                              again
         time *jt13 kn_cat "story"))

(m486!
 (act (m485 (action (m226 (lex say that)))
       (object
        (m484 (mode (m429 (lex should)))
         (object
          (m483 (min 0) (max 0)
           (arg
            (m482
             (act (m481 (action (m469 (lex require)))
                   (object
                    (m480 (forall v21)
                     (ant
                      (p127
                       (act (p126 (action (m260 (lex wish)))
                             (object
                              (p125
                               (act (m471 (action (m470 (lex pass)))
                                     (object b43)))
                               (agent v21)))))
                       (agent v21)))
                      (cq
                       (p129
                        (act (m479 (action (m246 (lex joust))) (against b43)))
                        (agent v21)))))))
              (agent b43)))))))))
 (agent b3) (kn_cat story) (time b45))

(m486!)

 CPU time : 0.02

*

;; Define "joust" as a verb
 ^(
-->defineVerb "joust")
"You want me to define the verb 'joust'.

I'll start by looking at the predicate stucture of the sentences I know that use 'joust'.  Here is

The most common type of sentences I know of that use 'joust' are of the form:
```

```
     'A something can joust.'

No superclasses were found for this verb.

 Sorting from the most common predicate case to the least common here is what I know.  I will firs

A basic ctgy can joust.

Now, looking from the bottom up I want to get a sense of the categories that most of the agents, c

A knight can joust.

"

 CPU time : 0.42

*
;; King Arthur jousts against the knight
(describe
 (assert agent *KA act
         (build action (build lex "joust") against *blknt2)
         time #jt17 kn_cat "story-comp"))

(m487! (act (m479 (action (m246 (lex joust))) (against b43))) (agent b3)
 (kn_cat story-comp) (time b46))

(m487!)

 CPU time : 0.00

*
;; The knight jousts against King Arthur
(describe
 (assert agent *blknt2 act
         (build action (build lex "joust") against *KA)
         time *jt17 kn_cat "story-comp"))

(m489! (act (m488 (action (m246 (lex joust))) (against b3))) (agent b43)
 (kn_cat story-comp) (time b46))

(m489!)

 CPU time : 0.01

*
;; King-Arthur fought the knight with some instrument
(describe
 (assert agent *KA act
         (build action (build lex "fight") object *blknt2 instr #KAspear1)
         time *jt17 kn_cat "story"))

(m492! (act (m491 (action (m490 (lex fight))) (instr b47) (object b43)))
 (agent b3) (kn_cat story) (time b46))

(m492!)

 CPU time : 0.01
```

```
*
;; The knight fought King Arthur with some other instrument
(describe
 (assert agent *blknt2 act
         (build action (build lex "fight") object *KA instr #knt2spear1)
         time *jt17 kn_cat "story"))

(m494! (act (m493 (action (m490 (lex fight))) (instr b48) (object b3)))
 (agent b43) (kn_cat story) (time b46))

(m494!)

 CPU time : 0.00

*
;; The fight took place after King Arthur and the knight met
(describe
(assert before *jt13 after *jt17 kn_cat "story-comp"))

(m495! (after b46) (before b45) (kn_cat story-comp))

(m495!)

 CPU time : 0.00

*
;; King Arthur's instrument is a spear
(describe
(assert member *KAspear1 class (build lex "spear") kn_cat "story"))

(m496! (class (m105 (lex spear))) (kn_cat story) (member b47))

(m496!)

 CPU time : 0.01

*
;; The knight's instrument is a spear
(describe
(assert member *knt2spear1 class (build lex "spear") kn_cat "story"))

(m497! (class (m105 (lex spear))) (kn_cat story) (member b48))

(m497!)

 CPU time : 0.00

*
;; Define joust as a verb
^(
-->defineVerb 'joust)
"You want me to define the verb 'joust'.

I'll start by looking at the predicate stucture of the sentences I know that use 'joust'.  Here is

The most common type of sentences I know of that use 'joust' are of the form:
     'A something can joust.'
```

```
No superclasses were found for this verb.

 Sorting from the most common predicate case to the least common here is what I know.  I will firs

A basic ctgy can joust.

Now, looking from the bottom up I want to get a sense of the categories that most of the agents, c

A king can joust.

"

 CPU time : 0.46

*
;; King Arthur grasped his something
(describe
 (assert agent *KA act
         (build action (build lex "grasp") object #KAsword time #jt14)
         kn_cat "story"))

(m504! (act (m503 (action (m502 (lex grasp))) (object b49) (time b50)))
 (agent b3) (kn_cat story))

(m504!)

 CPU time : 0.01

*
;; The thing King Arthur grasps is a sword
(describe
(assert member *KAsword class (build lex "sword") kn_cat "story"))

(m505! (class (m59 (lex sword))) (kn_cat story) (member b49))

(m505!)

 CPU time : 0.01

*
;; The spear fight took place before King Arthur grasped his sword
(describe
(assert before *jt17 after *jt14 kn_cat "story-comp"))

(m506! (after b50) (before b46) (kn_cat story-comp))

(m506!)

 CPU time : 0.00

*
;; The knight said that King Arthur should not draw his sword.
(describe
 (assert agent *blknt2 act
         (build action (build lex "say that")
                object (build mode (build lex "should")
                              object (build min 0 max 0
                                            arg (build agent *KA act
```

```
                                                    (build action (build lex "draw")
                                                          object *KAsword time #jt15)))))
         time *jt15 kn_cat "story"))

(m513!
 (act (m512 (action (m226 (lex say that)))
       (object
        (m511 (mode (m429 (lex should)))
         (object
          (m510 (min 0) (max 0)
           (arg
            (m509
             (act (m508 (action (m507 (lex draw))) (object b49) (time b51)))
             (agent b3)))))))))
 (agent b43) (kn_cat story) (time b51))

(m513!)

 CPU time : 0.01

*
;; The knight said that King Arthur should not draw his sword
;;  after King Arthur grasped his sword.
(describe
(assert before *jt14 after *jt15 kn_cat "story-comp"))

(m514! (after b51) (before b50) (kn_cat story-comp))

(m514!)

 CPU time : 0.01

*
;; The knight said that something was great.
(describe
 (assert agent *blknt2 act
         (build action (build lex "say that")
                object (build object #KAskill property (build lex "great")))
         time *jt15 kn_cat "story"))

(m517!
 (act (m516 (action (m226 (lex say that)))
       (object (m515 (object b52) (property (m376 (lex great)))))))
 (agent b43) (kn_cat story) (time b51))

(m517!)

 CPU time : 0.00

*
;; The thing that the knight said was great is
;;  King Arthur's skill
(describe
(assert object *KAskill rel (build lex "skill") possessor *KA kn_cat "story"))

(m519! (kn_cat story) (object b52) (possessor b3) (rel (m518 (lex skill))))

(m519!)
```

```
 CPU time : 0.00

*
;; King Arthur's skill is for jousting
(describe
(assert object1 *KAskill rel (build lex "purpose") object2 (build lex "joust")
        kn_cat "story"))

(m520! (kn_cat story) (object1 b52) (object2 (m246 (lex joust)))
 (rel (m285 (lex purpose))))

(m520!)

 CPU time : 0.01

*
;; The knight said that he wished to joust with King-Arthur again.
;;
(describe
 (assert agent *blknt2 act
         (build action (build lex "say that")
                object (build agent *blknt1 act
                              (build action (build lex "wish")
                                     object (build agent *blknt1 act
                                                   (build action (build lex "joust")
                                                          against *KA)
                                                   time #jt16))))
         time *jt15 kn_cat "story"))

(m525!
 (act (m524 (action (m226 (lex say that)))
       (object
        (m523
         (act (m522 (action (m260 (lex wish)))
               (object
                (m521 (act (m488 (action (m246 (lex joust))) (against b3)))
                 (agent b31) (time b53)))))
         (agent b31)))))
 (agent b43) (kn_cat story) (time b51))

(m525!)

 CPU time : 0.02

*
;; The black knight jousts with King Arthur after the
;;   knight says that he wants to joust again.
(describe
(assert before *jt15 after *jt16 kn_cat "story-comp"))

(m526! (after b53) (before b51) (kn_cat story-comp))

(m526!)

 CPU time : 0.01

*
```

```
;; Define "joust" as a verb
^(
-->defineVerb 'joust)
"You want me to define the verb 'joust'.

I'll start by looking at the predicate stucture of the sentences I know that use 'joust'.  Here is

The most common type of sentences I know of that use 'joust' are of the form:
     'A something can joust.'

No superclasses were found for this verb.

 Sorting from the most common predicate case to the least common here is what I know.  I will firs

A basic ctgy can joust.

Now, looking from the bottom up I want to get a sense of the categories that most of the agents, c

A king can joust.

"

 CPU time : 0.32

*
;; Then the chief lady of the castle said, 'Knight with the Two Swords,
;; ye must have ado and joust with a knight nearby who guards an island,
;; for no man may pass this way but he must joust ere he passes.'
;; 'That is an unhappy custom,' said Balyn, 'that a knight may not pass
;; this way unless he jousts.'
;; 'Ye shall have ado with only one knight,' said the lady.  [p. 58]
;;
;; A something said that something must joust with something else.
(describe
 (assert agent #jlady1 act
         (build action (build lex "say that")
                object (build mode (build lex "must")
                               object (build agent #Balyn act
                                             (build action (build lex "joust")
                                                    against #blknt3)
                                             time #jt18)))
        time #jt19 kn_cat "story"))

(m534!
 (act (m533 (action (m226 (lex say that)))
       (object
        (m532 (mode (m529 (lex must)))
         (object
          (m531 (act (m530 (action (m246 (lex joust))) (against b56)))
           (agent b55) (time b57)))))))
 (agent b54) (kn_cat story) (time b58))

(m534!)

 CPU time : 0.02

*
;; The speaker is a lady
```

```
(describe
(assert member *jlady1 class (build lex "lady") kn_cat "story"))

(m536! (class (m535 (lex lady))) (kn_cat story) (member b54))

(m536!)

 CPU time : 0.00

*
;; The thing that must joust is named "Balyn"
(describe
(assert object *Balyn proper-name (build lex "Balyn") kn_cat "story"))

(m538! (kn_cat story) (object b55) (proper-name (m537 (lex Balyn))))

(m538!)

 CPU time : 0.00

*
;; Balyn is a knight
(describe
(assert member *Balyn class (build lex "knight")
        kn_cat "story-comp"))

(m539! (class (m47 (lex knight))) (kn_cat story-comp) (member b55))

(m539!)

 CPU time : 0.01

*
;; The thing that Balyn must joust is a knight
(describe
(assert member *blknt3 class (build lex "knight") kn_cat "story"))

(m540! (class (m47 (lex knight))) (kn_cat story) (member b56))

(m540!)

 CPU time : 0.01

*
;; The ladys says that Balyn must joust before he actually jousts.
(describe
(assert before *jt19 after *jt18 kn_cat "story-comp"))

(m541! (after b57) (before b58) (kn_cat story-comp))

(m541!)

 CPU time : 0.00

*
;; The lady said that the knight guarded an island.
(describe
 (assert agent *jlady1 act
```

```
        (build action (build lex "say that")
               object (build agent *blknt3 act
                             (build action (build lex "guard")
                                    object #isle1)
                      time *jt19))
        time *jt19 kn_cat "story"))

(m546!
 (act (m545 (action (m226 (lex say that)))
       (object
        (m544 (act (m543 (action (m542 (lex guard))) (object b59)))
         (agent b56) (time b58)))))
 (agent b54) (kn_cat story) (time b58))

(m546!)

 CPU time : 0.01

*
;; The lady said that the knight required all men to joust with him,
;; if they wished to pass.
(describe
 (assert agent *jlady1 act
        (build action (build lex "say that")
               object (build forall *jknight
                             ant (build agent *jknight act
                                        (build action (build lex "wish")
                                               object (build agent *jknight act
                                                             (build action (build lex "pass")
                                                                    object *blknt3))))
                             cq (build agent *blknt3 act
                                       (build action (build lex "require")
                                              object (build agent *jknight act
                                                            (build action (build lex "joust")
                                                            against *blknt3))))) = oddact3)
        time *jt19 kn_cat "story"))

(m550!
 (act (m549 (action (m226 (lex say that)))
       (object
        (m548 (forall v21)
         (ant
          (p136
           (act (p135 (action (m260 (lex wish)))
                 (object
                  (p134 (act (m547 (action (m470 (lex pass))) (object b56)))
                   (agent v21)))))
           (agent v21)))
         (cq
          (p139
           (act (p138 (action (m469 (lex require)))
                 (object
                  (p137 (act (m530 (action (m246 (lex joust))) (against b56)))
                   (agent v21)))))
           (agent b56)))))))
 (agent b54) (kn_cat story) (time b58))

(m550!)
```

```
 CPU time : 0.01

*
;; Sir Balyn said that the knight requiring all men to joust with him,
;; if they wished to pass was something
(describe
 (assert agent *Balyn act
         (build action (build lex "say that")
                object (build member *oddact3 class #badcust))
         time #jt20 kn_cat "story"))

(m553!
 (act (m552 (action (m226 (lex say that)))
       (object
        (m551 (class b60)
         (member
            (m548 (forall v21)
             (ant
              (p136
               (act (p135 (action (m260 (lex wish)))
                     (object
                      (p134
                       (act (m547 (action (m470 (lex pass))) (object b56)))
                       (agent v21)))))
               (agent v21)))
             (cq
              (p139
               (act (p138 (action (m469 (lex require)))
                     (object
                      (p137
                       (act (m530 (action (m246 (lex joust))) (against b56)))
                       (agent v21)))))
               (agent b56)))))))))
 (agent b55) (kn_cat story) (time b61))

(m553!)

 CPU time : 0.02

*
;; The previous "something" is a custom
(describe
(assert subclass *badcust superclass (build lex "custom") kn_cat "story"))

(m555! (kn_cat story) (subclass b60) (superclass (m554 (lex custom))))

(m555!)

 CPU time : 0.00

*
;; The custom is unhappy
(describe
(assert object *badcust property (build lex "unhappy") kn_cat "story"))

(m557! (kn_cat story) (object b60) (property (m556 (lex unhappy))))
```

```
(m557!)

 CPU time : 0.00

*
;; Balyn speaks after the lady has spoken
(describe
(assert before *jt19 after *jt20 kn_cat "story-comp"))

(m558! (after b61) (before b58) (kn_cat story-comp))

(m558!)

 CPU time : 0.01

*
;; Define "joust" as a verb
^(
-->defineVerb 'joust)
"You want me to define the verb 'joust'.

I'll start by looking at the predicate stucture of the sentences I know that use 'joust'.  Here is

The most common type of sentences I know of that use 'joust' are of the form:
     'A something can joust.'

No superclasses were found for this verb.

 Sorting from the most common predicate case to the least common here is what I know.  I will firs

A basic ctgy can joust.

Now, looking from the bottom up I want to get a sense of the categories that most of the agents, c

A king can joust.

"

 CPU time : 0.56

*
;; '... therefore I wish to see you all together in the meadow of Camelot,
;; to joust and to tourney, so that after your death men may speak of it that
;; such good knights were here on such a day all together.'
;; To that counsel at the king's request they all accorded and put on their
;; armor suitable for jousting.  [p. 527]
;;
;; King Arthur said that he wished to see something joust.
;;
(describe
 (assert agent *KA act
        (build action (build lex "say that")
               object (build agent *KA act
                             (build action (build lex "wish")
                                    object (build agent *KA act
                                                  (build action (build lex "see")
                                                         object (build agent #Aknts act
                                                                       (build action (build lex "
```

```
                                                                    time #jt21))
                                                          time *jt21))
                                  time #jt22))
        time *jt22 kn_cat "story"))

(m567!
 (act (m566 (action (m226 (lex say that)))
       (object
        (m565
         (act (m564 (action (m260 (lex wish)))
               (object
                (m563
                 (act (m562 (action (m351 (lex see)))
                       (object
                        (m561 (act (m261 (action (m246 (lex joust)))))
                         (agent b62) (time b63)))))
                  (agent b3) (time b63)))))
         (agent b3) (time b64)))))
 (agent b3) (kn_cat story) (time b64))

(m567!)

 CPU time : 0.02

*
;; The thing that King Arthur wants to see joust is a knight.
(describe
(assert member *Aknts class (build lex "knight") kn_cat "story"))

(m568! (class (m47 (lex knight))) (kn_cat story) (member b62))

(m568!)

 CPU time : 0.01

*
;; The knights are King Arthur's knights
(describe
(assert object *Aknts rel (build lex "knight") possessor *KA kn_cat "story"))

(m569! (kn_cat story) (object b62) (possessor b3) (rel (m47 (lex knight))))

(m569!)

 CPU time : 0.00

*
;; King Arthur says that he wants to see the joust before it
;;   actually takes place
(describe
(assert before *jt22 after *jt21 kn_cat "story"))

(m570! (after b63) (before b64) (kn_cat story))

(m570!)

 CPU time : 0.00
```

```
*
;; Define joust as a verb
^(
-->defineVerb 'joust)
"You want me to define the verb 'joust'.

I'll start by looking at the predicate stucture of the sentences I know that use 'joust'.  Here is

The most common type of sentences I know of that use 'joust' are of the form:
     'A something can joust.'

No superclasses were found for this verb.

 Sorting from the most common predicate case to the least common here is what I know.  I will firs

A basic ctgy can joust.

Now, looking from the bottom up I want to get a sense of the categories that most of the agents, c

A king can joust.

"

 CPU time : 0.37

*
;; King-Arthur said that he wished to see his knights tourney.
(describe
 (assert agent *KA act
         (build action (build lex "say that")
                object (build agent *KA act
                              (build action (build lex "wish")
                                     object (build agent *KA act
                                                   (build action (build lex "see")
                                                          object (build agent *Aknts act
                                                                        (build action (build lex "
                                                                        time *jt21))
                                                          time *jt21))
                              time *jt22))
         time *jt22 kn_cat "story"))

(m579!
 (act (m578 (action (m226 (lex say that)))
       (object
        (m577
         (act (m576 (action (m260 (lex wish)))
               (object
                (m575
                 (act (m574 (action (m351 (lex see)))
                       (object
                        (m573 (act (m271 (action (m108 (lex tourney)))))
                         (agent b62) (time b63)))))
                 (agent b3) (time b63)))))
         (agent b3) (time b64)))))
 (agent b3) (kn_cat story) (time b64))

(m579!)
```

```
 CPU time : 0.02

*
;; King-Arthur's knights said that they would joust
(describe
 (assert agent *Aknts act
         (build action (build lex "say that")
                object (build agent *Aknts act
                              (build action (build lex "joust"))
                              time *jt21))
        time #jt23 kn_cat "story"))

(m581!
 (act (m580 (action (m226 (lex say that)))
       (object
        (m561 (act (m261 (action (m246 (lex joust))))) (agent b62)
         (time b63)))))
 (agent b62) (kn_cat story) (time b65))

(m581!)

 CPU time : 0.01

*
;; The joust takes place after King Arthur said that it
;;  would take place.
(describe
(assert before *jt23 after *jt21 kn_cat "story"))

(m582! (after b63) (before b65) (kn_cat story))

(m582!)

 CPU time : 0.01

*
;; The tourney takes place after King Arthur said
;;  that he wished to see it.
(describe
(assert before *jt22 after *jt23 kn_cat "story-comp"))

(m583! (after b65) (before b64) (kn_cat story-comp))

(m583!)

 CPU time : 0.00

*
;; The knights say that they will tourney
(describe
 (assert agent *Aknts act
         (build action (build lex "say that")
                object (build agent *Aknts act
                              (build action (build lex "tourney"))
                              time *jt21))
        time *jt23 kn_cat "story"))

(m585!
```

```
 (act (m584 (action (m226 (lex say that)))
       (object
        (m573 (act (m271 (action (m108 (lex tourney)))))) (agent b62)
         (time b63)))))
 (agent b62) (kn_cat story) (time b65))

(m585!)

 CPU time : 0.01

*
;; King Arthur's knights don something.
(describe
 (assert agent *Aknts act
         (build action (build lex "don")
                object #jarmor)
        time #jt24 kn_cat "story"))

(m588! (act (m587 (action (m586 (lex don))) (object b66))) (agent b62)
 (kn_cat story) (time b67))

(m588!)

 CPU time : 0.04

*
;; The thing they don is armor
(describe
(assert member *jarmor class (build lex "armor") kn_cat "story"))

(m590! (class (m589 (lex armor))) (kn_cat story) (member b66))

(m590!)

 CPU time : 0.00

*
;; The armor is for jousting
(describe
(assert object1 *jarmor rel (build lex "function") object2 (build lex "joust")
        kn_cat "story"))

(m592! (kn_cat story) (object1 b66) (object2 (m246 (lex joust)))
 (rel (m591 (lex function))))

(m592!)

 CPU time : 0.01

*
;; The knights joust
(describe
 (assert agent *Aknts act
         (build action (build lex "joust"))
         time *jt21 kn_cat "story-comp"))

(m593! (act (m261 (action (m246 (lex joust))))) (agent b62)
 (kn_cat story-comp) (time b63))
```

```
(m593!)

 CPU time : 0.01

*
;; The knights don their armor after they say they will tourney
(describe
(assert before *jt23 after *jt24 kn_cat "story-comp"))

(m594! (after b67) (before b65) (kn_cat story-comp))

(m594!)

 CPU time : 0.00

*
;; The knights don their armor before they joust
(describe
(assert before *jt24 after *jt21 kn_cat "story-comp"))

(m595! (after b63) (before b67) (kn_cat story-comp))

(m595!)

 CPU time : 0.01

*
;; Define "joust" as a verb
^(
--> defineVerb 'joust)
"You want me to define the verb 'joust'.

I'll start by looking at the predicate stucture of the sentences I know that use 'joust'.  Here is

The most common type of sentences I know of that use 'joust' are of the form:
     'A something can joust.'

No superclasses were found for this verb.

 Sorting from the most common predicate case to the least common here is what I know.  I will firs

A basic ctgy can joust.

Now, looking from the bottom up I want to get a sense of the categories that most of the agents, c

A king can joust.

"

 CPU time : 0.48

*
;; Define "joust" as a noun
^(
--> defineNoun 'joust)
 Definition of joust:
 Actions performed on a joust: king announce, baron arrange,
```

```
 Possible Properties: great, armor function,
nil

 CPU time : 0.40

*

End of /projects/stn2/CVA/demos/joust.demo demonstration.

 CPU time : 11.60

*
```

C Running the Algorithm

The following instructions for running SNePS and the noun-definition algorithm assume that the commands are being executed on a computer at the University at Buffalo Department of Computer Science and Engineering with access to the /projects filesystem.

- Type composer to start Allegro Common Lisp (or in Emacs type M-x run-acl).
- At the lisp prompt type (load "/projects/snwiz/bin/sneps"). Note: This algorithm has been tested with SNePS 2.6.0. The algorithm may or may not be compatible with previous versions of SNePS.
- Type (load "/projects/stn2/CVA/defun_noun.cl") to load the noun-definition algorithm.
- Type (sneps) to start SNePS.
- Type (demo "/projects/stn2/CVA/demos/some-demo.demo") to run a demo. Note: you should replace "some-demo" with the name of the demo you want to run.

D Location of Files

/projects/rapaport/CVA/STN2/defun_noun.cl
The noun definition algorithm.

/projects/rapaport/CVA/STN2/defun_verb.cl
The verb definition algorithm.

/projects/rapaport/CVA/STN2/demos
Contains all of the demos and supporting files.

/projects/rapaport/CVA/STN2/output
Contains sample runs of all the demos.

/projects/rapaport/CVA/STN2/papers
Contains PDF files of papers downloaded for CVA bibliography.

Online:
http://www.cse.buffalo.edu/ stn2/cva/index.html