Contextually Defining 'Gnomic'

and

Thoughts on Adjective Algorithms

Richard F. Doell

CSE 499 RAP

University at Buffalo

Abstract

This paper addresses the use of computational contextual vocabulary acquisition in finding a meaning for the adjective 'gnomic.' Computational contextual vocabulary is the use of algorithms to process a body of prior knowledge and a cotext, or the actual physical passage and unknown word, and extract a definition of that unknown word. The process was performed using the Semantic Network Processing System, or SNePS. A contextual definition of 'gnomic' was successfully found. Also explored were possibilities for an adjective algorithm and ways to move the process of finding the definition from a subconscious, or extra-conscious, method to a conscious one using the SNePS Rational Engine. These techniques are intended to come up with well defined ways of performing contextual vocabulary acquisition to then instruct human beings who may not have mastered the technique.

1 Introduction and Motivation

Contextual Vocabulary Acquisition, or CVA, is "the active, deliberate acquisition of a meaning for a word in a text by reasoning from textual clues and prior knowledge... It is the task faced by anyone coming upon an unknown word while reading, who has no outside source of help, but who needs to figure out a meaning for the word in order to understand the text being read" (Rapaport 2003). The context that is used in this process is a combination of the 'cotext,' or the words on the page or in the passage, and the prior knowledge of the reader (Rapaport 2003). This paper addresses an instance of Computational Contextual Vocabulary Acquisition, wherein a computational cognitive agent is substituted for a human in the CVA process.

The motivation for having a computer perform CVA is straightforward, but a bit of background may help the reader some. Many humans inherently have the ability to discern meaning from a context, but it is not a process which is taught and some students have difficulty mastering the ability. English as a second language students are especially good candidates for being explicitly taught the process for CVA in English, as are youth in English speaking countries. A case can be made that even some older students in America could benefit from developing the technique, given decreasing SAT and verbal skills across the country (Adams 2010). One of the large problems in teaching CVA explicitly is that there are few, if any, well-defined algorithms on how to do it. One paper (Clarke & Nation 1980) offers a series of steps, but culminates in 'guessing,' a process which humans are able to do, but, to a student who is already struggling with determining the meaning of a word, 'guessing' is often too vague of a process. If it were possible to determine an exact series of steps that should be performed in CVA, it could be taught to a human.

To develop this series of procedures, there are a number of likely options that appear. Asking people who are reading educators, existing 'CVA experts' and the like how they are able to perform CVA might be one path, but another presents itself upon closer examination of the problem. Given that there is some

procedure that could be used to extract information from context, including prior knowledge, the idea for using a computational agent should come to mind. From the early Church-Turing thesis, wherein, roughly, any algorithmically solvable problem can be solved by a Turing machine (Citation needed?), if there in fact does exist an algorithm for performing CVA, it could be done on a Turing machine. The 'Turing machine' that will be used in this instance of exploring computational CVA will be SNePS/Cassie, an intensional, propositional semantic network and its cognitive agent (Shapiro & Rapaport 1987).

Briefly, the SNePS semantic network uses an acyclic directed graph, composed of arcs and nodes, to represent knowledge about both the external world which exists physically and the mental world of ideas. A given concept or idea is represented by a node; relations between ideas are made up of molecular nodes, or nodes which have arcs coming from them. Those molecular nodes can represent both concepts, such as an act directed toward an object, to propositions portraying some feature of a real or imagined world. Propositions can be 'believed,' or asserted by the system, to represent things which are considered true. Different agents can also have beliefs; this is represented in the system as a belief about an agent having a belief. Using this model of a propositional digraph, incredibly complex information can be represented. The SNePS Inference Package, or SNIP, can be used to reason about those beliefs, 'unconsciously' traveling paths through the graph via various built-in logical rules, similar to first order predicate logic (Shapiro 1991).

If it is possible to create computable algorithms for performing CVA, it should be then possible to teach these methods to people who need or want to learn the CVA process. By teaching the cognitive agent, Cassie, how to perform CVA, we as researchers are able to make explicit that 'guess,' transforming a subconscious act into a conscious, active one. That idea, transforming computational CVA algorithms into a curriculum for teaching human beings, represents the long- term goal of the CVA project. This paper addresses a particular instance of teaching CVA to Cassie for an adjective.

2 'Gnomic' as an Unknown Adjective

To illustrate the process of CVA and to further the system of 'rules' which are used in CVA, the word 'gnomic' was defined by its context. The passage which was used to define the adjective is given below:

> "My criticisms echo an enigmatic passage from Post's posthumously published [diary]... Perhaps one should not read too much into this passage. Post's diary is notable for its highly fragmentary and elusive character, sometimes bordering on the mystical. ...Much of what Post says in the diary is extremely **gnomic**." (Rescorla 2007)

For those well versed in the definition of gnomic, this passage may hold quite the opposite properties of Post's diary, but for those who lack the ability it presents a difficulty. Here is where someone who is able to use contextual vocabulary acquisition would be able to take the elements in the cotext, such as the list of properties which describe the diary, for example, and extrapolate a meaning for gnomic from them. The difficulty arises when the reader is unfamiliar with those protocols and therefore cannot perform CVA. By default, Cassie is one of those readers.

The main components that facilitate CVA are the cotext, the background knowledge the reader has and the protocols which allow the reader to combine those things to create a definition. In the situation of the adjective, all that is available to the computational agent is the cotext. Background knowledge is not innate in computer programs, although attempts have been made at incorporating existing knowledge bases such as CYC and WordNet into SNePS to allow for automated acquisition of background knowledge (Dligach 2004). Algorithms that currently exist for computational CVA are well defined for nouns and adjectives, but are imprecise when it comes to adjectives (Rapaport & Kibby 2010). Therefore it is necessary to determine what background knowledge is appropriate and also to create some process for extracting a meaning for an adjective from context. The first task is by far the simpler one.

To get an idea of what background knowledge a typical reader would have in performing CVA, typical readers were asked. A note should be made, however, before the interview process is described. In CVA, much like solving a linear equation, it is only possible to find a 'good' meaning of a word if there is only one unknown in a context. With one word and one context, it is possible to 'solve' for a word. With multiple unknown words present, the process becomes much more difficult. The only solution would be to find one word's meaning in terms of another word, as in a system of linear equations. Given multiple contexts, it may be possible to determine multiple meanings, but the complexity increases greatly and, to simplify the task, only one unknown per context is permitted for the purposes of this paper. That said, as part of the interview process, informants were asked if any other words in the context were unknown; these words were then defined for the informant.

After the informant indicated that they knew all the words in a context, besides the unknown word, they were asked what they believed the unknown word to mean. In the case of gnomic, informants were read the passage, then asked for an initial impression. Once that impression, or 'definition,' was given, they were asked how they arrived at that meaning. General ideas, knowledge about the words: anything which was used to find the meaning was recorded. Various reports were given, even including a reference to the fantastic 'gnome,' but the trend among informants was that 'gnomic' meant some combination of the other adjectives in the sentence: fragmentary, elusive, mystical and enigmatic. The meanings of those words and the inference rules which led to a combination of them give a good base upon which to build the background knowledge for the reading of the passage.

3 Case Frames used in Representation

Once the prior knowledge had been established, the task remained as to how to represent it in SNePS. The existing CVA algorithms made use of a system of case frames to represent knowledge in a systematic

fashion, so, despite the fact that there is no extant adjective algorithm, it was decided that attempting to follow the current formalisms for knowledge representation. There are several case frames which are used in this representation.

The case frame which is used most extensively in the definition is the 'similar - similar' case frame. Although not one of the standard case frames, it is included for a number of reasons. While searching for dictionary definitions of the other adjectives in the cotext, it was noticed that most of the definitions were simply in terms of other adjectives. This would be useful in a 'weighted summation,' as networks of adjectives could be searched, but for the simple purposes of knowledge representation, something straightforward was needed. In the list of standard case frames, there exists a 'synonym-synonym' case frame. The problem with using this case frame is semantic in nature; the precise definition of a synonym is a bit vague when the network that is being utilized has facilities for having both synonyms of *words* and *ideas* which those words represent. To avoid this difficulty, the similar-similar case frame was invented to convey the idea that two concepts are similar, if not exact, and can be used as a drop-in for synonym-synonym.

The exact syntax of the format for these case frames is defined in SNePSUL, the SNePS user language. This language is defined elsewhere (Shapiro and SNePS Implementation Group 2010), but suffice it to say that it follows a Common Lisp-like prefix notation. The nodes created by SNePSUL commands follow the convention: any node with an exclamation point (!) is asserted/believed by the system, m represents a molecular node, b represents a base node (one which has arcs coming from it), v is a variable node used in pattern matching, and p is a molecular pattern node which is similar to one represented by an 'm,' but is connected to variables rather than constants. Molecular nodes can have any number of arcs coming from it, depending on the node. Now that this system has been made more explicit, depictions describing the different case frames used can be given with more meaning.

Figure 1 shows an example of the nodal representation of the similar-similar case frame. It depicts it

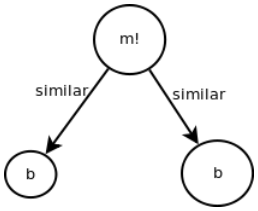

Figure 1: Similar-similar case frame, typically asserted.

with a base node as the endpoint of the arc. This is one possibility, but in the representation, it is much more likely to point to a molecular node which represents an idea expressed in English by a given word. Figure 2 shows the arc which is used to specify that case frame, 'lex.'

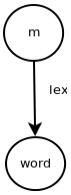

Figure 2: Lex case frame, represents the concept expressed as a word in English

Another of the case frames which is used to represent information about the class membership of an object is the member-class case frame. That, along with the subclass-superclass case frame, show the membership information of an object. These are depicted in Figures 3 and 4, respectively.

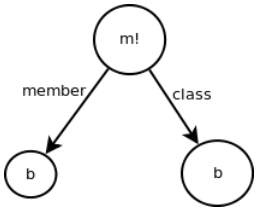

Figure 3: Member-class case frame, shows that an object (base node) is a member of some class

The next two case frames describe some property of an object. The first is the object-property case

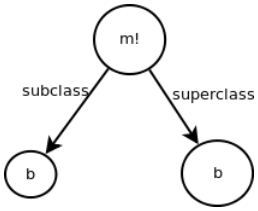

Figure 4: Subclass-superclass case frame, shows that a pair of classes are in the superclass subclass relation.

frame, which indicates that an object has some property; an adjective would be the property and the noun it describes would be the object. The second is the object proper-name case frame. This is useful in representing that an object is named. Post, the owner of the diary, is represented using one of these. All the case frames used are a part of the standard case frame library except for similar-similar.

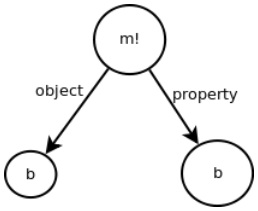

Figure 5: Object-property case frame. Describes properties of objects, like adjectives do

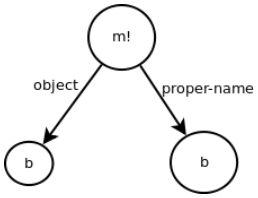

Figure 6: Object-proper name case frame. Describes the proper name of an object, like a state name or a person's name

The next case frame is used to represent ownership. The syntax might be odd to someone unfamiliar with the system, but the semantics are simple. Something stands in a relationship between two objects; one

is the possessor and the other is the possessed. The node created is depicted in Figure 7.

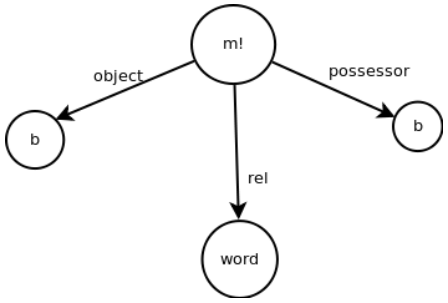

Figure 7: Object-rel-possessor case frame. Represents possession of the kind of thing represented by word of which the object is by the possessor.

Each of the previous case frames represents some belief about a thing, or an object. None of them give further information about how to connect the information or how it relates. SNePS implements the universal quantifier, $\forall$, quite nicely, and it is used to represent the rules that the informants gave. There are two types of for-all case frames which are used. Both can have any arbitrary number of variables, but the first relies on one antecedent and the latter has any number of conjunctive antecedents. Both have only one consequent. These are shown, in part, in Figures 8 and 9.

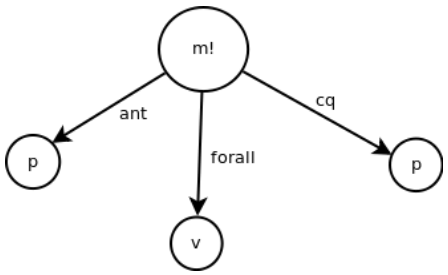

Figure 8: For all case frame. This is used to represent some universally quantified rule over the set of variables defined by the actual forall arc

These case frames all are used to represent the information and rules that the informants provided as

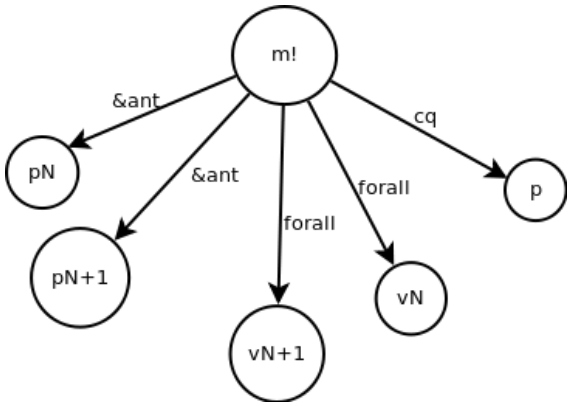

Figure 9: For all case frame with conjunct antecedents. Used whenever there are multiple criteria which must be matched for a rule to be used. N represents some number, which is appended to the letter in question, incrementing for every number of arguments and overall for the different kinds of nodes created.

part of their prior knowledge and the cotext itself, or a simplified version of it.

4 Description of Inference Rules

The background knowledge is made up of two primary parts: static knowledge, such as the fact that two words are 'similar,' and inference rules, such as the belief that if a string of adjectives describe a word, and one of them is unknown, then the unknown word is some sum of the others. When the cotext is 'read' by the computational agent - a process in which information is added to the knowledge base step by step - the different inference rules are triggered, adding more information to the knowledge base. Once this process of information adding and inference is complete, it is possible to query the knowledge base and the cognitive agent about what information it contains.

The rules that were used in the analysis and computational CVA of 'gnomic' were fairly simplistic semantically, if syntactically a bit complicated. In English, they can be explained as follows: If an object is

described by a series of adjectives, one of them being unknown, a meaning of the unknown adjective can be found by taking a weighted sum of the meanings of the other adjectives, if the list of other adjectives is equal to four in number. Different numbers of other adjectives will be examined later. The meaning of 'weighted sum,' is some combination of the meanings of the adjectives, stuck together in some fashion. What that translates into procedurally is a search through the similar words to the adjective list given. If four of them share the same 'meaning,' expressed as a similar word, then that similar word must have the same meaning as the unknown word, as it is the most general information available. The search then goes through each combination until the least general is found. The latter step is only taken for each set of adjectives where there is not already a given similar word. For example, the search on pairs of words sharing a similar word is not conducted if there already was a similar word found in the search for three sharing a common similar word. In practicality, this comes down to the pairs of given adjectives having similar words being dominant as there were not any similar words found between all four other adjectives in the sentence. This set of rules gives a fairly accurate definition of what the 'gnomic' means, at least as far as the informants were able to describe. The problem is that it only gives what words are similar to the adjective, rather than an actual 'definition' for an adjective, whatever that truly may be[1].

In order to provide more information for a definition, another set of rules were implemented. Informants were asked if they could give some basic class membership information about the unknown word. When polled, they said that the adjective was one which described a mental thing, which can be generalized into the realm of internal ideas. These contrast to external properties and adjectives, such as a thing's size or color. A basic set of rules were implemented to handle this functionality. For example: if a number of adjectives all belonged to the same class, then the unknown adjective also belonged to that class. Also, the subclass and superclass relations of the given adjective were used to place the unknown word into classes. If

[1]Some dictionaries do no more than this, that is, provide a list of near-synonyms for the given adjective

the known adjectives are members of a class which has a series of super classes, then the unknown adjective is likely also in that category. This facility was added to help overcome the problem of a definition strictly created by similar words or synonyms.

5 Results of Computational Contextual Vocabulary Acquisition With 'Gnomic'

When CVA was performed in SNePS using these rules and background info, the definition that was given was fairly accurate. When asked what words were similar, the system returned that gnomic was similar to: arcane, cryptic, puzzling, incomprehensible, mysterious and ambiguous. When compared to a dictionary definition of the word, this is not too far off. The class membership information, as predicted, attributed gnomic to the class of mental and internal ideas, which follows from the other adjectives being in those classes. Despite the lack of a formal algorithm with which to perform CVA with adjectives, it is fair to say that this project was successful in defining gnomic from context.

A sample run using SNePS 2.7.1, slightly edited for readability, can be found in Appendix B. The source code itself can be found in Appendix A.

6 Thoughts on an Adjective Algorithm, the Use of the Acting Engine and Future Work

The contextual analysis of 'gnomic' is largely complete. However, to further the overall task of computational contextual vocabulary acquisition, with particular attention to adjectives, there remain several problems which require more research. Some of the preliminary work on these can be seen in the previous section. One of the tasks which needs to be addressed is the implementation of the CVA algorithms in the acting engine of SNePS, the SNePS Rational Engine, SNeRE. The way that the current algorithms work

is not actually a cognitive process. The analogy that is often drawn with regards to this is that of opening up someone's brain and examining all the information that is present, drawing conclusions based on that information, then putting those conclusions into the brain. The point of re-writing the extant algorithms in SNeRE is to model more closely the thinking process. Simply developing the algorithms for CVA will further the project, but having a better cognitive model might further it more.

In the process of implementing this project, code was written which used SNeRE to implement the 'weighted sum' mentioned in previous sections. This was successful, and allowed for more control over the universally quantified statements. Implementing a form of sequence, selection and loop, SNeRE follows the principles of a programming language and is therefore slightly more intuitive in some cases. After an examination of the current algorithms and developing an understanding of the syntax and capabilities of SNeRE, it should not be particularly difficult to 'transcode' the algorithms. The primary reason that this was not accomplished as a part of this project was the time constraint and the need to come up with a process for extracting information about adjectives.

Another task which remains for future researchers is that of formalizing an adjective algorithm to parallel those which have been made for nouns and verbs. At the time of this project, there were several adjectives being processed. Although each did not follow precisely the same rules for CVA, they did have certain similarities. In addition to the 'weighted sum' algorithm for long lists of similar adjectives, there was another algorithm which worked for groups of adjectives all belonging to the same set. For a further examination of this algorithm, see the research of Yalei Song. Briefly, if there are adjectives describing the same thing, there is a chance that one will mean the complement of the other, in a binary relation, or the set difference of the rest, in a longer list scenario. These two algorithms might have some merit in a more general sense, but further research is required. To perform this research, it would be necessary to formulate surveys containing examples of naturally existing cotexts, likely to either support or contradict the claims of generality proposed

here. Once a general survey of the population has been completed, implementation of universal adjective algorithms could be attempted. At the very least, algorithms for very specific cases could be developed, such as in the case of several similar known adjectives describing the same object along with an unknown one.

Be it through SNeRE, an adjective algorithm or further analysis of different adjective cases, there is much work still to do with computational contextual vocabulary acquisition. Hopefully, future researchers will be able to formalize this theory of vocabulary acquisition and deliver it to potential candidates, be them foreign students or English speaking youth.

References

Adams, Marilyn Jager (2010-2011), "Advancing Our Students' Language and Literacy: The Challenge of Complex Texts", American Educator 34(4) (Winter): 311, 53.

Clarke, D.F., & Nation, I.S.P. (1980), "Guessing the Meanings of Words from Context: Strategy and Techniques", System 8: 211-220.

Dligach, Dmitriy (2004), "SNePS and WordNet: Using WordNet as a Source of Background Knowledge in Contextual Vocabulary Acquisition"

Rapaport, William J. & Kibby, Michael W. (submitted, 2010), "Contextual Vocabulary Acquisition: From Algorithm to Curriculum".

Rapaport, William J. (2003), "What Is the 'Context' for Contextual Vocabulary Acquisition?" (PDF), in Peter P. Slezak (ed.), Proceedings of the 4th International Conference on Cognitive Science/7th Australasian Society for Cognitive Science Conference (ICCS/ASCS-2003; Sydney, Australia) (Sydney: University of New South Wales), Vol. 2, pp. 547-552.

Rescorla, Michael (2007), "Church's Thesis and the Conceptual Analysis of Computability", Notre Dame Journal of Formal Logic 48(2): 253280; passages from pp. 265, 277n19, and 275, respectively.

Shapiro, S.C., and Rapaport, William J. (1987) “SNePS considered as a fully intensional propositional semantic network.” In N. Cercone and G. McCalla, editors, The Knowledge Frontier, pages 263-315. Springer-Verlag, New York, 1987.

Shapiro, S.C. and The SNePS Implementation Group, (2010) SNePS 2.7.1 User’s Manual, Department of Computer Science and Engineering, University at Buffalo, Buffalo, NY, September 3, 2010.

Stuart C. Shapiro. (1991) “Cables, paths and “subconscious” reasoning in propositional semantic networks.” In John F. Sowa, editor, Principles of Semantic Networks, chapter 4, pages 137-156. Morgan Kaufmann, San Mateo, CA, 1991.

Appendix A - Source Code of Gnomic CVA

```
; ========================================================================
; FILENAME:        gnomic-demo.txt
; DATE:               5_12_11
; PROGRAMMER:        Richard Doell

; Lines beginning with a semi-colon are comments.
; Lines beginning with "^" are Lisp commands.
; All other lines are SNePSUL commands.
;
; To use this file: run SNePS; at the SNePS prompt (*), type:
;
;        (demo "rdfoell-GNOMIC-demo.txt" :av)
;
; Make sure all necessary files are in the current working directory
; or else use full path names.
; ========================================================================

; Turn off inference tracing.
; This is optional; if tracing is desired, then delete this.
^(setq snip:*infertrace* nil)

;Using a custom 'algorithm' for adjective

; Clear the SNePS network:
(resetnet t)
```

```
; OPTIONAL:
; UNCOMMENT THE FOLLOWING CODE TO TURN FULL FORWARD INFERENCING ON:
;
 ;enter the "snip" package:
 ^(in-package snip)

; Leave ffi off
; ^(defun broadcast-one-report (represent)
;    (let (anysent)
;      (do.chset (ch *OUTGOING-CHANNELS* anysent)
;                (when (isopen.ch ch)
;                 (setq anysent
;                       (or (try-to-send-report represent ch)
;                           anysent)))))
;    nil)
;
 ;re-enter the "sneps" package:
 ^(in-package sneps)

; load all pre-defined relations:
; NB: If "intext" causes a "nil not of expected type" error,
;     then comment-out the "intext" command and then
;          uncomment & use the load command below, instead
^(load "/projects/rapaport/CVA/STN2/demos/rels")
;(intext "/projects/rapaport/CVA/STN2/demos/rels")



; load all pre-defined path definitions:
;(intext "/projects/rapaport/CVA/mkb3.CVA/paths/paths")
^(load "/projects/rapaport/CVA/mkb3.CVA/paths/paths")

; Define a 'similar' arc to avoid problems with synonym semantics
(define similar)
;this should define a path between asserted similar arcs
(define-path (similar- ! similar))

; BACKGROUND KNOWLEDGE:
; =====================
; (put annotated SNePSUL code of your background knowledge here)

;Putting in some of the similars for enigmatic, using thesaurus.com and
; the Merriam-Webster online thesaurus
(describe (assert similar (build lex "enigmatic") similar (build lex "ambiguous") similar
           (build lex "arcane") similar (build lex "cryptic") similar (build
```

```
lex "dark") similar (build lex "incomprehensible") similar (build lex "obscure") similar
(build lex "occult") similar (build lex "perplexing") similar (build lex "puzzling") similar
(build lex "mysterious")))

;Same, only for fragmentary
; Interestingly, fragmentary is the only one which has no similars among
; the other words
(describe (assert similar (build lex "fragmentary") similar (build lex "broken") similar (build lex
"incomplete") similar (build lex "disconnected") similar (build lex
"incoherent") similar (build lex "partial")))

;Now elusive
(describe (assert similar (build lex "elusive") similar (build lex "evasive") similar (build lex
"deceptive") similar (build lex "mysterious") similar (build lex "ambiguous") similar (build lex "incomprehensible")
similar (build lex "indefinable") similar (build lex "insubstantial") similar
*occult similar (build lex "puzzling")))

;Finally mystical
(describe (assert similar (build lex "mystical") similar *occult similar (build lex "arcane") similar (build lex "cryptic")
similar (build lex "esoteric")))

;Say that all the properties that directly describe the diary are mental concepts, things which are characteristics given
; by humans to things, ideas about things which don't exist outside the mind (giving a partial allowance to fragmentary for now,
;  also using the 'sense' of elusive and fragmentary which are more 'inside the mind' than not):
(describe (assert member ((build lex mystical) (build lex elusive) (build lex fragmentary) (build lex enigmatic)) class
(build lex mental)))

;Now say that mental ideas are subclasses of the intensional group of things

(describe (assert subclass (build lex mental) superclass (build lex
intensional)))

; This will take care of the extensional or intensional categorization
; that was used in previous demos as two split rules, instead; this
; follows any/all subclass/superclass paths and says that if four things
; all have the same superclass, then the fifth thing is a member of that
; class. This will be used with the subclass/superclass assertion above
; to say that gnomic is 'intensional' because it is a superclass of the
; idea of mental concepts/ideas, of which fragmentary, elusive, mystical
; and enigmatic are members of.
(describe (assert forall ($w $x $y $z $p $q $c)
        &ant(
        (build  member *p class unknown)
        (build  object *q property *w)
```

```
        (build  object *q property *x)
        (build  object *q property *y)
        (build  object *q property *z)
        (build  object *q property *p)
        (build  subclass ((find (class- member) *w)
                    (find (class- member) *x)
                    (find (class- member) *y)
                    (find (class- member) *z))
           superclass (build lex *c)))
        cq (build member *p class (build lex *c))))

; This is a slightly modified rule of the above, which says that if
; the adjectives are all of the same category, the unknown one is
; also in that category.

(describe (assert forall ($w $x $y $z $s $p $q)
        &ant(
        (build member *p class unknown)
        (build object *q property *w)
        (build object *q property *x)
        (build object *q property *y)
        (build object *q property *z)
        (build object *q property *p)
        (build member (*w *x *y *z) class *s))
        cq (build member *p class *s )))

; The idea for the following series of rules is as follows:
; If four words have something in common, then the fifth has
; that word in common.

;Then, if that doesn't trigger, check again for the more general case
; where three of the words have something in common. Then, if that doesn't
; happen, then trigger the rule for if two things have common. The idea is
; to get the mots general, and hopefully most correct, data about the unknown
; word. To implement this functionality, uncomment the bits about min/max

;; The following assertion says that we know no similar words to gnomic
;(describe (assert min 0 max 0 arg (build (similar (build lex gnomic)))))

; This rule states that if an object has four properties, and they
; all have a similar word to each of them, then the unknown word, the
; fifth property, is also similar to that word.
(describe (assert forall ($w $x $y $z $p $q)
           &ant(
```

```
                ; The properties all describe the same object
                ; Note: it's not actually possible to have all of these in one line-
                ; the original node which built them had to have the form of all arcs
                ; to one set.
                (build object *q property *p)
                (build object *q property *w)
                (build object *q property *x)
                (build object *q property *y)
                (build object *q property *z)
                ; p is unknown
                (build member *p class unknown))
          cq (
                build forall $s
                        ; The main properties all share a similar word
                        &ant(
                                (build similar *w similar *s)
                                (build similar *x similar *s)
                                (build similar *y similar *s)
                                (build similar *z similar *s))
                        cq (build similar *s similar *p))))

; This rule states that if an object has three properties, and they
; all have a similar word to each of them, then the unknown word, the
; fourth property, is also similar to that word.
(describe (assert forall ($x $y $z $p $q)
          &ant(
                ; The properties all describe the same object
                (build object *q property *p)
                (build object *q property *x)
                (build object *q property *y)
                (build object *q property *z)
                ;(build min 0 max 0 arg (build similar *p))
                ; p is unknown
                (build member *p class unknown))
          cq (
                build forall $s
                        ; The main properties all share a similar word
                        &ant(
                                (build similar *x similar *s)
                                (build similar *y similar *s)
                                (build similar *z similar *s))
                        cq (build similar *s similar *p))))

; This rule states that if an object has two properties, and they
```

```
; all have a similar word to each of them, then the unknown word, the
; third property, is also similar to that word.

(describe (assert forall ($x $y $p $q)
           &ant(
                ; The properties all describe the same object
                (build object *q property *p)
                (build object *q property *x)
                (build object *q property *y)
                ;(build min 0 max 0 arg (build similar *p))
                ; p is unknown
                (build member *p class unknown))
           cq (
                 build forall $s
                        ; The main properties all share a similar word
                       &ant(
                              (build similar *x similar *s)
                              (build similar *y similar *s))
                              ;(build similar *z similar *s))
                       cq (build similar *s similar *p))))


; CASSIE READS THE PASSAGE:
; =========================
; (put annotated SNePSUL code of the passage here)

; Post has a proper name of 'Post'
(describe (add object #Post proper-name (build lex "Post")))
; A diary is owned by Post
(describe (add object #diary rel (build lex "diary") possessor *Post))
; The diary is a diary
(describe (add member *diary class (build lex "diary")))
; The diary is enigmatic
(describe (add object *diary property (build lex "enigmatic")))
; The diary is fragmentary
(describe (add object *diary property (build lex "fragmentary")))
; The diary is elusive
(describe (add object *diary property (build lex "elusive")))
; The diary is mystical
(describe (add object *diary property (build lex "mystical")))

;The diary is 'gnomic,' which is a member of the
; class of unknown things
(describe (add object *diary property (build lex "gnomic")))
```

```
; Saying that the word gnomic is unknown

(describe (add member (build lex "gnomic") class unknown))

; Ask Cassie what "gnomic" means:

;First, ask what classes gnomic is in:
(describe (find (class- ! member) (build lex gnomic)))

; Then, get the words which are 'similar' to it:
(describe (findassert similar (build lex gnomic)))
```

Appendix B - Demo Run

```
/Sneps-Linux-Exe-2.7.1$ ./sneps.sh
International Allegro CL Enterprise Edition
8.2 [Linux (x86)] (Sep 16, 2010 13:53)
Copyright (C) 1985-2010, Franz Inc., Oakland, CA, USA.  All Rights Reserved.

This standard runtime copy of Allegro CL was built by:
   [4549] University at Buffalo

;;---
;; Current reader case mode: :case-sensitive-lower
SNePS-2.7 [PL:1 2009/03/24 14:53:32] loaded.
Type `(sneps)' or `(snepslog)' to get started.
cl-user(1): (sneps)
```

```
   Welcome to SNePS-2.7 [PL:1 2009/03/24 14:53:32]

Copyright (C) 1984--2007 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.

   5/12/2011 13:13:46

* (demo "demo/CVA/rfdoell-GNOMIC-demo.txt")

File CVA/rfdoell-GNOMIC-demo.txt is now the source of input.


 CPU time : 0.00

* ; ===========================================================================
; FILENAME:        gnomic-demo.txt
; DATE:               5_12_11
; PROGRAMMER:        Richard Doell

; Lines beginning with a semi-colon are comments.
```

```
; Lines beginning with "^" are Lisp commands.
; All other lines are SNePSUL commands.
;
; To use this file: run SNePS; at the SNePS prompt (*), type:
;
;         (demo "rdfoell-GNOMIC-demo.txt" :av)
;
; Make sure all necessary files are in the current working directory
; or else use full path names.
; ============================================================================


; Turn off inference tracing.
; This is optional; if tracing is desired, then delete this.
^(
--> setq snip:*infertrace* nil)
nil




 CPU time : 0.00


*
;Using a custom 'algorithm' for adjective
```

```
; Clear the SNePS network:
(resetnet t)

Net reset


 CPU time : 0.00


*
; OPTIONAL:
; UNCOMMENT THE FOLLOWING CODE TO TURN FULL FORWARD INFERENCING ON:
;
 ;enter the "snip" package:
 ^(
--> in-package snip)
#<The snip package>



 CPU time : 0.00


*
```

```
; Leave ffi off
; ^(defun broadcast-one-report (represent)
;    (let (anysent)
;      (do.chset (ch *OUTGOING-CHANNELS* anysent)
;                (when (isopen.ch ch)
;                 (setq anysent
;                       (or (try-to-send-report represent ch)
;                           anysent)))))
;    nil)
;
 ;re-enter the "sneps" package:
 ^(
--> in-package sneps)
#<The sneps package>




 CPU time : 0.00


*
; load all pre-defined relations:
; NB: If "intext" causes a "nil not of expected type" error,
;     then comment-out the "intext" command and then
```

```
;          uncomment & use the load command below, instead
^(
--> load "/projects/rapaport/CVA/STN2/demos/rels")
t



 CPU time : 0.01

* ;(intext "/projects/rapaport/CVA/STN2/demos/rels")



; load all pre-defined path definitions:
;(intext "/projects/rapaport/CVA/mkb3.CVA/paths/paths")
^(
--> load "/projects/rapaport/CVA/mkb3.CVA/paths/paths")
before implied by the path (compose before
                            (kstar (compose after- ! before)))
before- implied by the path (compose (kstar (compose before- ! after))
                             before-)
after implied by the path (compose after
                           (kstar (compose before- ! after)))
after- implied by the path (compose (kstar (compose after- ! before))
```

```
                              after-)
sub1 implied by the path (compose object1- superclass- ! subclass
                            superclass- ! subclass)
sub1- implied by the path (compose subclass- ! superclass subclass- !
                            superclass object1)
super1 implied by the path (compose superclass subclass- ! superclass
                            object1- ! object2)
super1- implied by the path (compose object2- ! object1 superclass- !
                             subclass superclass-)
superclass implied by the path (or superclass super1)
superclass- implied by the path (or superclass- super1-)
t



 CPU time : 0.00


*
; Define a 'similar' arc to avoid problems with synonym semantics
(define similar)


(similar)
```

```
 CPU time : 0.00

* ;this should define a path between asserted similar arcs
(define-path (similar- ! similar))


 CPU time : 0.00

*
; BACKGROUND KNOWLEDGE:
; =====================
; (put annotated SNePSUL code of your background knowledge here)

;Putting in some of the similars for enigmatic, using thesaurus.com and
; the Merriam-Webster online thesaurus
(describe (assert similar (build lex "enigmatic") similar
           (build lex "ambiguous") similar
           (build lex "arcane") similar (build lex "cryptic") similar
           (build lex "dark") similar (build lex "incomprehensible") similar
           (build lex "obscure") similar
(build lex "occult") similar
        (build lex "perplexing") similar
        (build lex "puzzling") similar
```

```
(build lex "mysterious")))

(m12!
 (similar (m11 (lex mysterious)) (m10 (lex puzzling))
  (m9 (lex perplexing)) (m8 (lex occult)) (m7 (lex obscure))
  (m6 (lex incomprehensible)) (m5 (lex dark)) (m4 (lex cryptic))
  (m3 (lex arcane)) (m2 (lex ambiguous)) (m1 (lex enigmatic))))

(m12!)

 CPU time : 0.02

*
;Same, only for fragmentary
; Interestingly, fragmentary is the only one which has no similars among
; the other words
(describe (assert similar (build lex "fragmentary") similar
           (build lex "broken") similar (build lex
"incomplete") similar (build lex "disconnected") similar (build lex
"incoherent") similar (build lex "partial")))

(m19!
 (similar (m18 (lex partial)) (m17 (lex incoherent))
```

```
  (m16 (lex disconnected)) (m15 (lex incomplete)) (m14 (lex broken))
  (m13 (lex fragmentary))))

(m19!)

 CPU time : 0.00

*
;Now elusive
(describe (assert similar (build lex "elusive") similar
           (build lex "evasive") similar (build lex
"deceptive") similar
           (build lex "mysterious") similar
           (build lex "ambiguous") similar
           (build lex "incomprehensible")
        similar (build lex "indefinable") similar
        (build lex "insubstantial") similar
        *occult similar (build lex "puzzling")))

(m25!
 (similar (m24 (lex insubstantial)) (m23 (lex indefinable))
  (m22 (lex deceptive)) (m21 (lex evasive)) (m20 (lex elusive))
  (m11 (lex mysterious)) (m10 (lex puzzling))
```

```
  (m6 (lex incomprehensible)) (m2 (lex ambiguous))))

(m25!)

 CPU time : 0.00

*
;Finally mystical
(describe (assert similar (build lex "mystical")
           similar *occult similar (build lex "arcane") similar
           (build lex "cryptic")
        similar (build lex "esoteric")))

(m28!
 (similar (m27 (lex esoteric)) (m26 (lex mystical)) (m4 (lex cryptic))
  (m3 (lex arcane))))

(m28!)

 CPU time : 0.00

*
;Say that all the properties that directly describe the diary are
```

```
        mental concepts, things which are characteristics given
; by humans to things, ideas about things which don't exist outside the mind
        (giving a partial allowance to fragmentary for now,
;  also using the 'sense' of elusive and fragmentary which are more
        'inside the mind' than not):
(describe (assert member ((build lex mystical)
        (build lex elusive) (build lex fragmentary)
        (build lex enigmatic)) class
(build lex mental)))

(m30! (class (m29 (lex mental)))
 (member (m26 (lex mystical)) (m20 (lex elusive))
   (m13 (lex fragmentary)) (m1 (lex enigmatic))))

(m30!)

 CPU time : 0.00

*
;Now say that mental ideas are subclasses of the intensional group of things

(describe (assert subclass (build lex mental) superclass (build lex
intensional)))
```

```
(m32! (subclass (m29 (lex mental)))
 (superclass (m31 (lex intensional))))

(m32!)

 CPU time : 0.01

*
; This will take care of the extensional or intensional categorization
; that was used in previous demos as two split rules, instead; this
; follows any/all subclass/superclass paths and says that if four things
; all have the same superclass, then the fifth thing is a member of that
; class. This will be used with the subclass/superclass assertion above
; to say that gnomic is 'intensional' because it is a superclass of the
; idea of mental concepts/ideas, of which fragmentary, elusive, mystical
; and enigmatic are members of.
(describe (assert forall ($w $x $y $z $p $q $c)
        &ant(
        (build  member *p class unknown)
        (build  object *q property *w)
        (build  object *q property *x)
        (build  object *q property *y)
```

```
        (build  object *q property *z)
        (build  object *q property *p)
        (build  subclass ((find (class- member) *w)
                          (find (class- member) *x)
                          (find (class- member) *y)
                          (find (class- member) *z))
           superclass (build lex *c)))
        cq (build member *p class (build lex *c))))

(m33! (forall v7 v6 v5 v4 v3 v2 v1)
 (&ant (p8 (superclass (p7 (lex v7)))) (p6 (object v6) (property v5))
  (p5 (object v6) (property v4)) (p4 (object v6) (property v3))
  (p3 (object v6) (property v2)) (p2 (object v6) (property v1))
  (p1 (class unknown) (member v5)))
 (cq (p9 (class (p7)) (member v5))))

(m33!)

 CPU time : 0.00

*
; This is a slightly modified rule of the above, which says that if
; the adjectives are all of the same category, the unknown one is
```

```
; also in that category.


(describe (assert forall ($w $x $y $z $s $p $q)
        &ant(
        (build member *p class unknown)
        (build object *q property *w)
        (build object *q property *x)
        (build object *q property *y)
        (build object *q property *z)
        (build object *q property *p)
        (build member (*w *x *y *z) class *s))
        cq (build member *p class *s )))


(m34! (forall v14 v13 v12 v11 v10 v9 v8)
 (&ant (p16 (class v12) (member v11 v10 v9 v8))
  (p15 (object v14) (property v13)) (p14 (object v14) (property v11))
  (p13 (object v14) (property v10)) (p12 (object v14) (property v9))
  (p11 (object v14) (property v8)) (p10 (class unknown) (member v13)))
 (cq (p17 (class v12) (member v13))))


(m34!)


 CPU time : 0.00
```

```
*
; The idea for the following series of rules is as follows:
; If four words have something in common, then the fifth has
; that word in common.

; This rule states that if an object has four properties, and they
; all have a similar word to each of them, then the unknown word, the
; fifth property, is also similar to that word.
(describe (assert forall ($w $x $y $z $p $q)
           &ant(
                ; The properties all describe the same object
                ; Note: it's not actually possible to have
                        all of these in one line-
                ; the original node which built
                        them had to have the form of all arcs
                ; to one set.
                (build object *q property *p)
                (build object *q property *w)
                (build object *q property *x)
                (build object *q property *y)
                (build object *q property *z)
                ; p is unknown
```

```
                (build member *p class unknown))

          cq (

                build forall $s

                       ; The main properties all share a similar word

                      &ant(

                             (build similar *w similar *s)

                             (build similar *x similar *s)

                             (build similar *y similar *s)

                             (build similar *z similar *s))

                      cq (build similar *s similar *p))))


(m35! (forall v20 v19 v18 v17 v16 v15)
 (&ant (p23 (class unknown) (member v19))
  (p22 (object v20) (property v18)) (p21 (object v20) (property v17))
  (p20 (object v20) (property v16)) (p19 (object v20) (property v15))
  (p18 (object v20) (property v19)))
 (cq
  (p29 (forall v21)
   (&ant (p27 (similar v21 v18)) (p26 (similar v21 v17))
    (p25 (similar v21 v16)) (p24 (similar v21 v15)))
   (cq (p28 (similar v21 v19))))))


(m35!)
```

```
 CPU time : 0.00


*
; This rule states that if an object has three properties, and they
; all have a similar word to each of them, then the unknown word, the
; fourth property, is also similar to that word.
(describe (assert forall ($x $y $z $p $q)
           &ant(
                ; The properties all describe the same object
                (build object *q property *p)
                (build object *q property *x)
                (build object *q property *y)
                (build object *q property *z)
                ; p is unknown
                (build member *p class unknown))
           cq (
                 build forall $s
                        ; The main properties all share a similar word
                       &ant(
                              (build similar *x similar *s)
                              (build similar *y similar *s)
                              (build similar *z similar *s))
```

```
                    cq (build similar *s similar *p))))

(m36! (forall v26 v25 v24 v23 v22)
 (&ant (p34 (class unknown) (member v25))
  (p33 (object v26) (property v24)) (p32 (object v26) (property v23))
  (p31 (object v26) (property v22)) (p30 (object v26) (property v25)))
 (cq
  (p39 (forall v27)
   (&ant (p37 (similar v27 v24)) (p36 (similar v27 v23))
    (p35 (similar v27 v22)))
   (cq (p38 (similar v27 v25))))))

(m36!)

 CPU time : 0.00

*
; This rule states that if an object has two properties, and they
; all have a similar word to each of them, then the unknown word, the
; third property, is also similar to that word.

(describe (assert forall ($x $y $p $q)
           &ant(
```

```
                ; The properties all describe the same object
                (build object *q property *p)
                (build object *q property *x)
                (build object *q property *y)
                ; p is unknown
                (build member *p class unknown))
          cq (
                build forall $s
                        ; The main properties all share a similar word
                       &ant(
                              (build similar *x similar *s)
                              (build similar *y similar *s))
                              ;(build similar *z similar *s))
                       cq (build similar *s similar *p))))


(m37! (forall v31 v30 v29 v28)
 (&ant (p43 (class unknown) (member v30))
  (p42 (object v31) (property v29)) (p41 (object v31) (property v28))
  (p40 (object v31) (property v30)))
 (cq
  (p47 (forall v32)
   (&ant (p45 (similar v32 v29)) (p44 (similar v32 v28)))
   (cq (p46 (similar v32 v30))))))
```

```
(m37!)

 CPU time : 0.00

*

; CASSIE READS THE PASSAGE:
; ========================
; (put annotated SNePSUL code of the passage here)

; Post has a proper name of 'Post'
(describe (add object #Post proper-name (build lex "Post")))

(m39! (object b1) (proper-name (m38 (lex Post))))

(m39!)

 CPU time : 0.01

* ; A diary is owned by Post
(describe (add object #diary rel (build lex "diary") possessor *Post))
```

```
(m41! (object b2) (possessor b1) (rel (m40 (lex diary))))

(m41!)

 CPU time : 0.00

* ; The diary is a diary
(describe (add member *diary class (build lex "diary")))

(m42! (class (m40 (lex diary))) (member b2))

(m42!)

 CPU time : 0.00

* ; The diary is enigmatic
(describe (add object *diary property (build lex "enigmatic")))

(m46! (superclass (m31 (lex intensional))))
(m43! (object b2) (property (m1 (lex enigmatic))))
(m30! (class (m29 (lex mental)))
 (member (m26 (lex mystical)) (m20 (lex elusive))
   (m13 (lex fragmentary)) (m1)))
```

```
(m46! m43! m30!)

 CPU time : 0.24

* ; The diary is fragmentary
(describe (add object *diary property (build lex "fragmentary")))

(m47! (object b2) (property (m13 (lex fragmentary))))

(m47!)

 CPU time : 0.07

* ; The diary is elusive
(describe (add object *diary property (build lex "elusive")))

(m48! (object b2) (property (m20 (lex elusive))))

(m48!)

 CPU time : 0.07
```

```
* ; The diary is mystical
(describe (add object *diary property (build lex "mystical")))

(m49! (object b2) (property (m26 (lex mystical))))

(m49!)

 CPU time : 0.07

*
;The diary is 'gnomic,' which is a member of the
; class of unknown things
(describe (add object *diary property (build lex "gnomic")))

(m51! (object b2) (property (m50 (lex gnomic))))

(m51!)

 CPU time : 0.07

*
; Saying that the word gnomic is unknown
```

```
(describe (add member (build lex "gnomic") class unknown))


(m158! (forall v32)
 (&ant (p97 (similar v32 (m20 (lex elusive))))
  (p95 (similar v32 (m26 (lex mystical)))))
 (cq (p96 (similar (m50 (lex gnomic)) v32))))
(m157! (forall v32)
 (&ant (p98 (similar v32 (m13 (lex fragmentary)))) (p95)) (cq (p96)))
(m156! (forall v32) (&ant (p98) (p97)) (cq (p96)))
(m155! (forall v27)
 (&ant (p93 (similar v27 (m13))) (p91 (similar v27 (m20)))
  (p90 (similar v27 (m26))))
 (cq (p92 (similar (m50) v27))))
(m148! (similar (m27 (lex esoteric)) (m3 (lex arcane))))
(m147! (similar (m27) (m4 (lex cryptic))))
(m146! (similar (m3) (m2 (lex ambiguous))))
(m145! (similar (m5 (lex dark)) (m3)))
(m144! (similar (m6 (lex incomprehensible)) (m3)))
(m143! (similar (m7 (lex obscure)) (m3)))
(m142! (similar (m8 (lex occult)) (m3)))
(m141! (similar (m9 (lex perplexing)) (m3)))
(m140! (similar (m10 (lex puzzling)) (m3)))
(m139! (similar (m11 (lex mysterious)) (m3)))
```

```
(m138! (similar (m4) (m2)))

(m137! (similar (m4) (m3)))

(m136! (similar (m5) (m4)))

(m135! (similar (m6) (m4)))

(m134! (similar (m7) (m4)))

(m133! (similar (m8) (m4)))

(m132! (similar (m9) (m4)))

(m131! (similar (m10) (m4)))

(m130! (similar (m11) (m4)))

(m121! (similar (m50) (m2)))

(m120! (similar (m50) (m11)))

(m119! (similar (m50) (m6)))

(m118! (similar (m50) (m10)))

(m117! (similar (m20) (m2)))

(m116! (similar (m20) (m6)))

(m115! (similar (m20) (m10)))

(m114! (similar (m20) (m11)))

(m113! (similar (m21 (lex evasive)) (m20)))

(m112! (similar (m22 (lex deceptive)) (m20)))

(m111! (similar (m23 (lex indefinable)) (m20)))

(m110! (similar (m24 (lex insubstantial)) (m20)))

(m109! (similar (m50) (m3)))

(m108! (similar (m50) (m4)))
```

```
(m107! (similar (m26) (m3)))

(m106! (similar (m26) (m4)))

(m105! (similar (m27) (m26)))

(m104! (similar (m14 (lex broken)) (m13)))

(m103! (similar (m15 (lex incomplete)) (m13)))

(m102! (similar (m16 (lex disconnected)) (m13)))

(m101! (similar (m17 (lex incoherent)) (m13)))

(m100! (similar (m18 (lex partial)) (m13)))

(m99! (similar (m2) (m1 (lex enigmatic))))

(m98! (similar (m3) (m1)))

(m97! (similar (m4) (m1)))

(m96! (similar (m5) (m1)))

(m95! (similar (m6) (m1)))

(m94! (similar (m7) (m1)))

(m93! (similar (m8) (m1)))

(m92! (similar (m9) (m1)))

(m91! (similar (m10) (m1)))

(m90! (similar (m11) (m1)))

(m61! (forall v32) (&ant (p98) (p94 (similar v32 (m1)))) (cq (p96)))

(m60! (forall v32) (&ant (p97) (p94)) (cq (p96)))

(m59! (forall v32) (&ant (p95) (p94)) (cq (p96)))

(m58! (forall v27) (&ant (p93) (p91) (p89 (similar v27 (m1))))
 (cq (p92)))
```

```
(m57! (forall v27) (&ant (p93) (p90) (p89)) (cq (p92)))
(m56! (forall v27) (&ant (p91) (p90) (p89)) (cq (p92)))
(m55! (forall v21)
 (&ant (p87 (similar v21 (m20))) (p86 (similar v21 (m13)))
  (p85 (similar v21 (m26))) (p84 (similar v21 (m1))))
 (cq (p88 (similar (m50) v21))))
(m54! (class (m29 (lex mental))) (member (m50)))
(m53! (class (m31 (lex intensional))) (member (m50)))
(m52! (class unknown) (member (m50)))
(m51! (object b2) (property (m50)))
(m49! (object b2) (property (m26)))
(m48! (object b2) (property (m20)))
(m47! (object b2) (property (m13)))
(m43! (object b2) (property (m1)))

(m158! m157! m156! m155! m148! m147! m146! m145! m144! m143! m142!
 m141! m140! m139! m138! m137! m136! m135! m134! m133! m132! m131!
 m130! m121! m120! m119! m118! m117! m116! m115! m114! m113! m112!
 m111! m110! m109! m108! m107! m106! m105! m104! m103! m102! m101!
 m100! m99! m98! m97! m96! m95! m94! m93! m92! m91! m90! m61! m60! m59!
 m58! m57! m56! m55! m54! m53! m52! m51! m49! m48! m47! m43!)

 CPU time : 5.49
```

```
*
; Ask Cassie what "gnomic" means:

;First, ask what classes gnomic is in:
(describe (find (class- ! member) (build lex gnomic)))

(m29 (lex mental))
(m31 (lex intensional))
(unknown)

(m29 m31 unknown)

 CPU time : 0.00

*
; Then, get the words which are 'similar' to it:
(describe (findassert similar (build lex gnomic)))

(m108! (similar (m50 (lex gnomic)) (m4 (lex cryptic))))
(m109! (similar (m50) (m3 (lex arcane))))
(m118! (similar (m50) (m10 (lex puzzling))))
(m119! (similar (m50) (m6 (lex incomprehensible))))
```

```
(m120! (similar (m50) (m11 (lex mysterious))))

(m121! (similar (m50) (m2 (lex ambiguous))))


(m108! m109! m118! m119! m120! m121!)


 CPU time : 0.00


*


End of CVA/rfdoell-GNOMIC-demo.txt demonstration.



 CPU time : 6.06


* (lisp)

"End of SNePS"

cl-user(2): :ex
```