Computing the Meaning of ‘kolper’ using SNePS-based CVA Algorithms

Rahul Krishna

krishna2@cse.buffalo.edu

16 Dec., 2004

CSE 740: Seminar on Contextual Vocabulary Acquisition

Abstract

Contextual Vocabulary Acquisition (CVA) is an interdisciplinary research project between the Department of Computer Science and Engineering, and the Department of Learning and Instruction at the University at Buffalo. The goal of this project is to simulate computationally the process of acquisition of word meanings through context and to use this computational model in formulating a curriculum that will teach school children how to make use of contextual information to acquire the meanings of unfamiliar words. The CVA project is mainly concerned with formulating algorithms that will determine the meaning of some word in a sentence given adequate background knowledge about the meanings of all the other words in that sentence. The SNePS Knowledge Representation and Reasoning system is used to represent the appropriate background knowledge, and the CVA algorithm is then used to infer the meaning of an unfamiliar word. This paper reports on the use of CVA techniques to infer the meaning of the noun ‘kolper’. The results are presented and directions for future research are also outlined.

1. Contextual Vocabulary Acquisition (CVA): An Overview

Contextual Vocabulary Acquisition (Rapaport & Ehrlich, 2000) is a research project between the Department of Computer Science and Engineering, and the Department of Learning and Instruction at the University at Buffalo. The goal of this project is to develop a computational theory of vocabulary acquisition and to integrate the findings of this research to formulate a curriculum that can be taught to students in school. The researchers at the Department of Learning and Instruction are involved in the educational component of this research project. The researchers at the Department of Computer Science and Engineering are involved in formulating algorithms that infer the meaning of an unknown word in a sentence using contextual clues.

Context is used to mean the surrounding text, grammatical information, morphological information, and background knowledge. Currently, the CVA algorithm uses only surrounding text and background knowledge to infer the meaning of an unfamiliar word. The process by which the meaning of an unknown word is inferred is as follows. First, verbal think-aloud protocols are collected to determine what background knowledge is required for humans to infer the meaning of an unfamiliar word. Second, this background knowledge is represented using the SNePS knowledge representation and reasoning system. Next, the sentence with the unknown word is represented in SNePS. Finally, a CVA algorithm is run so as to infer the meaning of an unfamiliar word. Currently, there are two CVA algorithms – the noun algorithm and the verb algorithm - which are used to infer the meanings of unfamiliar nouns and verbs respectively.

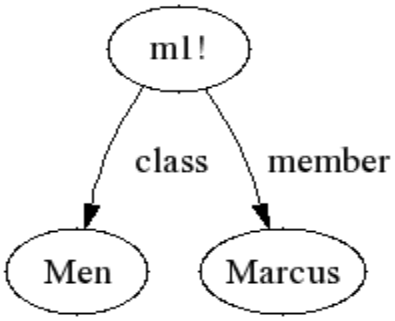
2. Introduction to SNePS

SNePS (Shapiro & Rapaport, 1987) is a knowledge representation and reasoning system that can represent knowledge about real and imaginary worlds and make inferences based on the knowledge that is represented. The SNePS system consists of an embedded 'humanoid' agent called 'Cassie'. SNePS is a form of a semantic network to model Cassie's mind and cognition. Classical semantic networks are graphs where the nodes represent concepts and the arcs represent the relations between concepts. However, in the classical semantic networks, propositions about propositions cannot be represented by graphs. SNePS solves this problem by using a propositional semantic network where the nodes represent propositions and the arcs just provide structural information.

SNePS does not come with pre-defined relations except for those involved in reasoning. Instead, the user can define his or her own relations based on the necessity of the domain to be represented. For example, suppose we want to represent: 'Marcus is a Man'. This can be done using the following statement

(assert member Marcus class Men)

which says that Marcus is a member of the class of Men. The graphical representation for the above proposition is



SNePS consists of many packages and features that make it useful for the CVA research project – SNIP (SNePS Inference Package), SNePSUL (SNePS User Language), and SNeBR (SNePS Belief Revision Package). SNIP is the package that does the reasoning. It allows for two types of reasoning – path based and node based reasoning. SNePSUL is a user interface that allows users to build the networks. SNeBR is used for belief revision when the SNePS system encounters a contradiction in its knowledge base and some of the beliefs have to be revised.

3. Word and Context

The sentence with an unknown word was taken from Van Daalen-Kapteijns & Elshout-Mohr (1981: 387)

> When you're used to a broad view it becomes quite depressing when you come to live in a room with one or two kolpers fronting a courtyard.

The above passage was used in a study (described in Van Daalen-Kapteijns & Elshout-Mohr, 1981) in which the authors report on a series of experiments that they conducted to gain insight into how the meaning of an unfamiliar word is learnt from context. The subjects of the experiments were asked to define the meaning of the word kolper through the use of context. The noun "kolper" is an imaginary word and was constructed just for the purposes of this experiment.

In this project, we represent using SNePS, the background knowledge required to infer the meaning of “kolper”, and use the CVA noun algorithm to have Cassie compute the meaning of “kolper”.

4. Verbal Protocols

The first step in the project is deciding what the program should output as the meaning of the word. This will have a bearing on what background knowledge would have to be represented for the program to accomplish the above. Two separate sources of information were used in deciding what this goal should be.

First, Van Daalen-Kapteijns & Elshout-Mohr (1981) outline an ideal acquisition process in which they hypothesize that the meaning that must be inferred based on this sentence alone is that a kolper is a type of window.

> Replacing ‘kolpers’ by ‘windows’ would result in a comprehensible sentence, so the provisional representation for ‘kolper’ will be modeled after that of ‘window’, with slots reserved for more specific information to differentiate kolpers from other windows. (Van Daalen-Kapteijns & Elshout-Mohr, 1981: 387).

Second, we conducted our own tests with people to see what they thought was the meaning of kolper. The general consensus was that a kolper was a type of window that did not provide a view.

So, here we have two similar goals. In the first, the kolper is a type of window and thus inherits all the default properties of windows, which includes the property of providing a view. An implicit assumption regarding the default properties of windows made here is that when people think of windows, they think of a view rather than something that blocks a view. A window is an object that in most cases provides a view. For the second goal, according to the other verbal protocols we conducted, a kolper is again a type of window, but here the definition is qualified by the observation that a kolper is a kind of window that does not provide a view.

Two separate demos were written to model these two goals. Of course, the two demos do not differ greatly and the second may be seen as an extension to the first. Nevertheless, both the demos are presented here, and the background knowledge required for each is explained. We will refer to the first demo as Demo 1 and to the second demo as Demo 2.

5. Knowledge Representation in SNePS

5.1. Case Frames

All case frames except the mod/head case frame were adapted from Napieralski's standard CVA case frames (Napieralski, 2003). The syntax and semantics of the mod/head case frame is given in Appendix A.

5.2. Background Knowledge

Demo 1

The following rules were added to represent the background knowledge

A room has windows

This is the rule that brings the concept of windows into the background knowledge. It defines the relation between rooms and windows. The sentence contains the word “room” but there is no mention anywhere of a window. This rule helps Cassie connect room with windows.

```
(describe (assert forall $room
     ant (build member *room class (build lex "room"))
     cq (build member
          (build skolem-function window\ of arg1 *room)
           *class (build lex "window"))
     cq (build object #win rel
          (build skolem-function window\ of arg1
           *room) possessor *room)))
```

Skolem functions are used in the place of existential quantifiers. SNePS no longer provides existential quantifier(Martins, 2000 : 185). So what the above rules says is that, for every room, the object that is the window of that particular room is possessed by the room. The object-rel-possessor case frame was used as opposed to the part-whole case frame because the former is more general than the latter. Consider the sentence ‘The room has gringlers’[1]. Now gringlers could be taken to mean something that is part of the room. But, given that we don’t know what gringlers mean, it could also be taken to mean, say, some

alien creatures. While alien creatures cannot be said to be part of the room, the room can

---

1. Dr. William J. Rapaport, personal communication

definitely possess them. The rationale for this choice can be seen more clearly below in the rule which says that a room possesses kolpers, where we do not know what “kolpers” means.

A window provides a view

This following rule describes the relation between a view, which is mentioned in the sentence, and a window. The connection between the room and window is provided in the rule above.

```
(describe (assert forall $window
     ant (build member *window class (build lex "window"))
     cq (build object1 *window rel (build lex "provides")
          object2
          (build skolem-function view\ of arg1 *window))
     cq (build member
          (build skolem-function view\ of arg1 *window)
          class (build lex "view"))))
```

A courtyard is outside of a room

This rule connects two other words that occur in the sentence – “courtyard” and “room”. This rule asserts the spatial relation between a courtyard and a room. This rule will be used as an antecedent in the next rule.

```
(describe (assert forall ($cyard $room)
     &ant (build member *cyard class
          (build lex "courtyard"))
     &ant (build member *room class (build lex "room"))
     cq (build object1 *cyard rel (build lex "outside")
          object2 *room)))
```

If an object X fronts something that is outside a room and object X is possessed by a room then X provides a view

This rule is required to tell Cassie about what it means to front a courtyard and what are the implications of some object fronting a courtyard.

```
(describe (assert forall ($x $y $k $room)
     &ant (build object1 *x rel (build lex "front")
               object2 *y)
     &ant (build object1 *y rel (build lex "outside")
               object2 *room)
     &ant (build member *room class (build lex "room"))
     &ant (build possessor *room object *x rel *k)
     cq (build object1 *x rel (build lex "provides")
               object2
               (build skolem-function view\ of arg1 *x))
     cq (build member
               (build skolem-function view\ of arg1 *x)
          class (build lex "view"))))
```

If X is Y's Z then X is a member of class Z

This is a defeasible rule that says that if an object x bears the relationship of Z with Y, then X is a member of the class denoted by Z. This can be understood better in the light of an example. If we have the sentence ‘Mary is John’s sister’, then Mary is a member of the class of sisters. The rule is used in this demo so that Cassie can infer that the object that is a kolper of some room is a member of the class of kolpers.

```
(describe (assert forall ($x $y $z)
     ant (build object *x possessor *y rel *z)
     cq (build member *x class *z)))
```

If objects X & Y provide views, and X & Y are members of some classes A & B respectively, and the class A is unknown, then class A is a sub-class of class B

This is another defeasible rule. This rule is the rule that connects kolpers and windows. This rule does not refer to any universal truth, but we hypothesize that it is these kind of defeasible rules that humans employ while performing tasks of a similar nature.

```
(describe (assert forall ($a $b $w $x $y $z)
     &ant (build object1 *x rel (build lex "provides")
               object2 *w)
     &ant (build object1 *y rel (build lex "provides")
               object2 *z)
     &ant (build member *x class *a)
     &ant (build member *y class *b)
     &ant (build member *w class (build lex "view"))
     &ant (build member *z class (build lex "view"))
     &ant (build object *a property (build lex "unknown"))
     cq (build subclass *a superclass *b)))
```

These were the rules that were added as background knowledge. We will now discuss the background knowledge that was used for the second demo, after which we'll talk about how the sentence was represented and what the results were.

Demo 2

The following background knowledge was used for the second demo.

A room has windows

This rule is defined in the same way as in Demo 1.

A courtyard is outside of a room

This rule is defined in the same way as in Demo 1.

A window possibly provides a view

This is where there is a change in the background knowledge as compared with Demo 1. Here we make explicit to Cassie that a window providing a view is not a universal truth but is possibly true. While in Demo 1, providing a view was a property of windows, here it is only a possible property.

```
(describe (assert forall $window
     ant (build member *window class (build lex "window"))
     cq (build mod (build lex "possibly") head
          (build object1 *window rel (build lex "provides")
           object2
           (build skolem-function view\ of arg1 *window)))
     cq (build member
          (build skolem-function view\ of arg1 *window)
               class (build lex "view"))))
```

A window possibly does not provide a view

We explicitly mention that the opposite of the previous rule, i.e., the window might possibly not provide a view. In writing the previous rule and this rule, we make use of the mod-head case frame. The reason for using this is to avoid having SNeBR (the SNePS Belief Revision System) detect a contradiction.

```
(describe (assert forall $window
     ant (build member *window class (build lex "window"))
     cq (build mod (build lex "possibly") head
          (build min 0 max 0 arg
          (build object1 *window rel (build lex "provides")
          object2
          (build skolem-function view\ of arg1 *window))))
     cq (build member
          (build skolem-function view\ of arg1 *window)
               class (build lex "view"))))
```

If an object X fronts something that is outside a room and object X is possessed by a room then X possibly provides a view

This rule is similar to its counterpart in Demo 1 but in the light of the previous two rules, the a modification has to be made to the rule used in Demo 1. Here we say that the object X that fronts something and is possessed by a room possibly provides a view.

```
(describe (assert forall ($x $y $k $room)
     &ant (build object1 *x rel (build lex "front")
               object2 *y)
     &ant (build object1 *y rel (build lex "outside")
               object2 *room)
     &ant (build member *room class (build lex "room"))
     &ant (build possessor *room object *x rel *k)
     cq (build mod (build lex "possibly") head
          (build object1 *x rel (build lex "provides")
               object2
               (build skolem-function view\ of arg1 *x)))
     cq (build member
          (build skolem-function view\ of arg1 *x)
               class (build lex "view"))))
```

If an object X fronts something that is outside a room and object X is possessed by a room then X possibly does not provide a view

This rule complements the previous rule. That is, the object X that fronts something and is possessed by a room possibly does not provide a view.

```
(describe (assert forall ($x $y $k $room)
     &ant (build object1 *x rel (build lex "front")
               object2 *y)
     &ant (build object1 *y rel (build lex "outside")
               object2 *room)
     &ant (build member *room class (build lex "room"))
     &ant (build possessor *room object *x rel *k)
     cq (build mod (build lex "possibly") head
          (build min 0 max 0 arg
            (build object1 *x rel (build lex "provides")
               object2
```

```
            (build skolem-function view\ of arg1 *x))))
     cq (build member
          (build skolem-function view\ of arg1 *x) class
               (build lex "view"))))
```

If X is Y's Z then X is a member of class Z

This rule is the same as used in the Demo 1.

```
(describe (assert forall ($x $y $z)
     ant (build object *x possessor *y rel *z)
     cq (build member *x class *z)))
```

If objects X & Y possibly provide views, and X & Y are members of some classes A & B respectively, and the class A is unknown, then class A is a sub-class of class B

This rule is similar to its counterpart in Demo 1 with the modification of possibly providing a view.

```
(describe (assert forall ($a $b $w $x $y $z)
     &ant (build mod (build lex "possibly") head
          (build object1 *x rel (build lex "provides")
               object2 *w))
     &ant (build mod (build lex "possibly") head
          (build object1 *y rel (build lex "provides")
               object2 *z))
     &ant (build member *x class *a)
     &ant (build member *y class *b)
     &ant (build member *w class (build lex "view"))
     &ant (build member *z class (build lex "view"))
     &ant (build object *a property (build lex "unknown"))
     cq (build subclass *a superclass *b)))
```

If X is used to a view and X is depressed and a view is possibly provided or not provided by some Z, then y is not provided by that z

This is the key connecting piece of background knowledge that helps Cassie infer that a kolper does not provide a view. It connects the concepts of being used to something and being depressed when the person does not get something he or she is used to.

```
(describe (add forall ($x $v1 $v2 $z $view)
     &ant (build object *x property
               (build lex "depressed"))
     &ant (build object1 *x rel (build lex "used-to")
               object2 *v1)
     &ant (build member *v1 class *view)
     &ant (build mod (build lex "possibly") head
               (build object1 *z rel (build lex "provides")
                    object2 *v2))
     &ant (build mod (build lex "possibly") head
               (build min 0 max 0 arg
                    (build object1 *z rel (build lex
                    "provides") object2 *v2)))
     &ant (build member *v2 class *view)
     cq (build min 0 max 0 arg
          (build object1 *z rel (build lex "provides")
               object2 *v2))))
```

If we have a rule such that for any X and Y, if we have the negation of (X provides Y), this rule is equivalent to the rule (X does not provide Y)

This rule was added to take care of a current limitation of the CVA algorithm. In looking for possible properties, the CVA algorithm looks only for assertions that have not been negated. Thus, this rule was added so that Cassie detects the ‘does not provide view’ property of kolpers.

```
(describe (add forall ($x $y)
     ant (build min 0 max 0 arg (build object1 *x
               rel (build lex "provides") object2 *y))
     cq (build object1 *x rel
               (build lex "does not provide") object2 *y)))
```

5.3. Representing the Sentence

> When you're used to a broad view it becomes quite depressing when you come to live in a room with one or two kolpers fronting a courtyard. (Van Daalen-Kapteijns and Elshout-Mohr, 1981: 387).

The above sentence has the structure of an if-then statement. That is, the above sentence must be represented as a 'forall' rule with antecedents and consequents. The antecedents must be about an agent being used to a broad view and living in a room that has kolpers and in which the kolpers front a courtyard. The consequent must be that the agent is depressed. That is, if the antecedents are true, then the agent must be depressed.

```
(describe (add forall ($view $agent $room $kolper
                        $courtyard)
     &ant (build member *view class (build lex "view"))
     &ant (build object *view property (build lex "broad"))
     &ant (build object1 *agent rel (build lex "used to ")
               object2 *view)
     &ant (build agent *agent act
               (build action (build lex "live")
               object *room))
     &ant (build member *room class (build lex room))
     &ant (build possessor *room object *kolper
               rel (build lex "kolper"))
     &ant (build object1 *kolper rel (build lex "front")
               object2 *courtyard)
     &ant (build member *courtyard
               class (build lex "courtyard"))
     cq (build object *agent
               property (build lex "depressed"))))
```

Furthermore, it was noticed that, since the above is represented as a 'forall' rule, it wasn't really specifying facts in the sense that Cassie expects. That is, there was no explicit knowledge about any instances that satisfied the above rule. Thus, one specific instance

had to be provided. The following assertions were added, and these assertions would trigger the above node-based inference rule.

There is a specific view.

```
(add member #view class (build lex "view"))
```

That view is broad

```
(add object *view property (build lex "broad"))
```

There is some agent that lives in a particular place.

```
(add agent #fred act (build action (build lex "live")
     object #room))
```

That agent is used to the view.

```
(add object1 *fred rel (build lex "used-to") object2 *view)
```

The place where the agent lives is a room.

```
(add member *room class (build lex "room"))
```

The room contains a kolper.

```
(add possessor *room object #kolper rel
     (build lex "kolper"))
```

The kolper fronts some object.

```
(add object1 *kolper rel (build lex "front")
     object2 #courtyard)
```

The object which the kolper fronts is a courtyard.

```
(add member *courtyard class (build lex "courtyard"))
```

Kolper has the property unknown

```
(add object (build lex "kolper")
     property (build lex "unknown"))
```

The reason for using ‘add’ instead of ‘assert’ to represent the sentence is to trigger forward inference in SNePS when the sentence is being read. This is to emulate a human reader’s inference processes while reading a sentence.

6. Cassie’s definition of kolper

The following definitions were computed by Cassie for each of the demos.

Demo 1:

```
; Ask Cassie what "kolper" means:
^(
--> defineNoun "kolper")
 Definition of kolper:
 Class Inclusions: window,
 Possible Properties: front courtyard, provides view,
provides view,
 Possessive: room,
nil
```

Demo 2:

```
; Ask Cassie what "kolper" means:
^(
--> defineNoun "kolper")
 Definition of kolper:
 Class Inclusions: window,
 Possible Properties: does not provide view, does not
provide view, front courtyard,
 Possessive: room,
nil
```

It can be noted that Cassie's definitions are what were expected, given the background knowledge. The first demo produced the result as stated in Van Daalen-Kapteijns and Elshout-Mohr(1981). The second demo produced the results in accordance with the verbal protocols we conducted.

7. Future Work

7.1. Short Term Goals:

One of the immediate short term goals is to see the effect of the word 'broad' used in the sentence. Neither Van Daalen-Kapteijns and Elshout-Mohr(1981) nor any of the people we interviewed attached any specific significance to the word 'broad' in the process of inference of the meaning of kolper. Nonetheless, it would be interesting to model computationally the effect of 'broad' by adding some background knowledge about the meaning of 'broad' and examining Cassie's output.

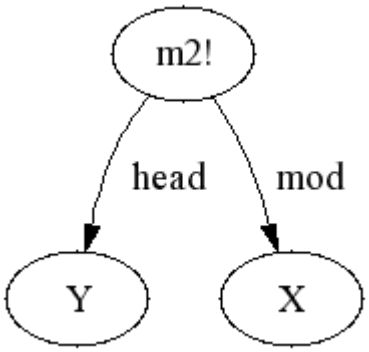
7.2. Long Term Goals:

The sentence worked on in this project is only one of five sentences that Van Daalen-Kapteijns and Elshout-Mohr(1981) used in their study of the process of acquisition of word meanings. In their paper, they use “kolper” in five sentences, and each sentence emphasizes different aspect of the meaning of “kolper”. The subjects of the experiments are expected to incrementally update their model of the meaning of “kolper” on reading each sentence. So, one of the long term goals of this research would be to represent the background knowledge required for all the sentences, and have Cassie incrementally learn the meaning of the word. For example, Demo 2 returns ‘does not provide view’ as a possible property. After all the sentences have been read, it should be clear that kolpers do not provide a view, and the ‘does not provide view’ property should be promoted from ‘possible property’ to a ‘full property’, i.e., an inherent property of all kolpers. Moreover, since Van Daalen-Kapteijns and Elshout-Mohr give quite a detailed analysis of the acquisition process in humans, it would be very interesting to see how the computer model resembles or differs from the human model.

Van Daalen-Kapteijns and Elshout-Mohr strike an analogy between the process of acquisition of word meanings in humans and the process of how concepts were acquired by Winston’s arch-learning program (Winston, 1977). In Winston’s work, the program was given pictures of a concept (for example, an arch), and the program would develop a structural description and incrementally update it as it was shown more images. It would be interesting to develop a computational model of the acquisition of word meanings based on Winston’s approach to learning concepts.

Van Daalen-Kapteijns and Elshout-Mohr, in their study, also talk about the differences in the models that were used by the human subjects to acquire word meanings. Specifically, they speak of two models – analytical and holistic. Another direction of research for this project could be to explore how these models would be represented computationally and how this would affect the acquisition of word meanings.

Appendix A: mod/head case frame



Semantics:

[[m2]] is the proposition that [[X]] is a modifier to the head phrase [[Y]]

Example:

John is possibly sick

(build agent #john property (build head sick mod possibly))

Appendix B: Run for Demo 1

```
* ;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
; FILENAME: kolper.demo.1
; DATE:           10/29/2004 - 11/23/2004
; PROGRAMMER:     Rahul Krishna

;; this template version:     template.demo.2003.11.17.txt

; Lines beginning with a semi-colon are comments.
; Lines beginning with "^" are Lisp commands.
; All other lines are SNePS commands.
;
; To use this file: run SNePS; at the SNePS prompt (*), type:
;
;     (demo "kolper.demo" :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

*
; Load the appropriate definition algorithm:
^(
--> load "/projects/rapaport/CVA/STN2/defun_noun.cl")
; Loading /projects/rapaport/CVA/STN2/defun_noun.cl
t



 CPU time : 0.21

*
; Clear the SNePS network:
(resetnet t)

Net reset


 CPU time : 0.01

*
; 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

* ;
; ;turn on full forward inferencing:
 ^(
--> 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)
broadcast-one-report



 CPU time : 0.00

*
 ;re-enter the "sneps" package:
 ^(
--> in-package sneps)
#<The sneps package>



 CPU time : 0.00

*
; load all pre-defined relations:
(intext "/projects/rapaport/CVA/STN2/demos/rels")
File /projects/rapaport/CVA/STN2/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 : 0.02
```

```
*

End of file /projects/rapaport/CVA/STN2/demos/rels


 CPU time : 0.02

*
; load all pre-defined path definitions:
(intext "/projects/rapaport/CVA/mkb3.CVA/paths/paths")
File /projects/rapaport/CVA/mkb3.CVA/paths/paths is now the source of
input.


 CPU time : 0.00

*
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

*
sub1 implied by the path (compose object1- superclass- ! subclass
superclass-
                          ! subclass)
sub1- implied by the path (compose subclass- ! superclass subclass- !
                           superclass object1)


 CPU time : 0.00

*
super1 implied by the path (compose superclass subclass- ! superclass
object1-
                            ! object2)
super1- implied by the path (compose object2- ! object1 superclass- !
subclass
                             superclass-)


 CPU time : 0.00

*
```

```
superclass implied by the path (or superclass super1)
superclass- implied by the path (or superclass- super1-)


 CPU time : 0.00

*

End of file /projects/rapaport/CVA/mkb3.CVA/paths/paths


 CPU time : 0.01

*
; Define some other relations
(define skolem-function arg1)

(skolem-function arg1)

 CPU time : 0.00

*
; BACKGROUND KNOWLEDGE:
; =====================

; A courtyard is outside of a room.
(describe (assert forall ($cyard $room)
      &ant (build member *cyard class (build lex "courtyard"))
      &ant (build member *room class (build lex "room"))
      cq (build object1 *cyard rel (build lex "outside") object2
*room)))

(m4! (forall v2 v1)
 (&ant (p2 (class (m2 (lex room))) (member v2))
  (p1 (class (m1 (lex courtyard))) (member v1)))
 (cq (p3 (object1 v1) (object2 v2) (rel (m3 (lex outside))))))

(m4!)

 CPU time : 0.00

*
; A room has windows
(describe (assert forall *room
      ant (build member *room class (build lex "room"))
      cq (build member (build skolem-function window\ of arg1 *room)
class (build lex "window"))
      cq (build object #win rel (build skolem-function window\ of arg1
*room) possessor *room)))

(m6! (forall v2) (ant (p2 (class (m2 (lex room))) (member v2)))
 (cq
  (p6 (object b1) (possessor v2)
   (rel (p4 (arg1 v2) (skolem-function window of))))
  (p5 (class (m5 (lex window))) (member (p4)))))

(m6!)
```

```
 CPU time : 0.00

*
; A windows provides a view
(describe (assert forall $window
      ant (build member *window class (build lex "window"))
      cq (build object1 *window rel (build lex "provides") object2
(build skolem-function view\ of arg1 *window))
      cq (build member (build skolem-function view\ of arg1 *window)
class (build lex "view"))))

(m9! (forall v3) (ant (p7 (class (m5 (lex window))) (member v3)))
 (cq
  (p10 (class (m8 (lex view)))
   (member (p8 (arg1 v3) (skolem-function view of))))
  (p9 (object1 v3) (object2 (p8)) (rel (m7 (lex provides))))))

(m9!)

 CPU time : 0.00

*
;if an object fronts something that is outside a room and object is
possessed by the room then it provides a view
(describe (assert forall ($x $y $k $room)
      &ant (build object1 *x rel (build lex "front") object2 *y)
      &ant (build object1 *y rel (build lex "outside") object2 *room)
      &ant (build member *room class (build lex "room"))
      &ant (build possessor *room object *x rel *k)
      cq (build object1 *x rel (build lex "provides") object2 (build
skolem-function view\ of arg1 *x))
      cq (build member (build skolem-function view\ of arg1 *x) class
(build lex "view"))))

(m11! (forall v7 v6 v5 v4)
 (&ant (p14 (object v4) (possessor v7) (rel v6))
  (p13 (class (m2 (lex room))) (member v7))
  (p12 (object1 v5) (object2 v7) (rel (m3 (lex outside))))
  (p11 (object1 v4) (object2 v5) (rel (m10 (lex front)))))
 (cq
  (p17 (class (m8 (lex view)))
   (member (p15 (arg1 v4) (skolem-function view of))))
  (p16 (object1 v4) (object2 (p15)) (rel (m7 (lex provides))))))

(m11!)

 CPU time : 0.00

*
; if x is y's z then x is a member of class z
(describe (assert forall ($x $y $z)
      ant (build object *x possessor *y rel *z)
      cq (build member *x class *z)))

(m12! (forall v10 v9 v8) (ant (p18 (object v8) (possessor v9) (rel
v10)))
```

```
 (cq (p19 (class v10) (member v8))))

(m12!)

 CPU time : 0.00

*
;if x and y both provide views, and x & y are members of classes a & b
repectively, and class a is unknown, then class a is a subclass of
class b
(describe (assert forall ($a $b $w $x $y $z)
      &ant (build object1 *x rel (build lex "provides") object2 *w)
      &ant (build object1 *y rel (build lex "provides") object2 *z)
      &ant (build member *x class *a)
      &ant (build member *y class *b)
      &ant (build member *w class (build lex "view"))
      &ant (build member *z class (build lex "view"))
      &ant (build object *a property (build lex "unknown"))
      cq (build subclass *a superclass *b)))

(m14! (forall v16 v15 v14 v13 v12 v11)
 (&ant (p26 (object v11) (property (m13 (lex unknown))))
  (p25 (class (m8 (lex view))) (member v16)) (p24 (class (m8)) (member
v13))
  (p23 (class v12) (member v15)) (p22 (class v11) (member v14))
  (p21 (object1 v15) (object2 v16) (rel (m7 (lex provides))))
  (p20 (object1 v14) (object2 v13) (rel (m7))))
 (cq (p27 (subclass v11) (superclass v12))))

(m14!)

 CPU time : 0.00

*


; CASSIE READS THE PASSAGE:
; =========================

; The Sentence:
; When you are used to a broad view, it becomes quite depressing
; when you come to live in a room with one or two kolpers
; fronting a courtyard.

(describe (add forall ($view $agent $room $kolper $courtyard)
      &ant (build member *view class (build lex "view"))
      &ant (build object *view property (build lex "broad"))
      &ant (build object1 *agent rel (build lex "used to ") object2
*view)
      &ant (build agent *agent act (build action (build lex "live")
object *room))
      &ant (build member *room class (build lex room))
      &ant (build possessor *room object *kolper rel (build lex
"kolper"))
      &ant (build object1 *kolper rel (build lex "front") object2
*courtyard)
      &ant (build member *courtyard class (build lex "courtyard"))
```

```
      cq (build object *agent property (build lex "depressed"))))

(m20! (forall v21 v20 v19 v18 v17)
 (&ant (p36 (class (m1 (lex courtyard))) (member v21))
  (p35 (object1 v20) (object2 v21) (rel (m10 (lex front))))
  (p34 (object v20) (possessor v19) (rel (m18 (lex kolper))))
  (p33 (class (m2 (lex room))) (member v19))
  (p32 (act (p31 (action (m17 (lex live))) (object v19))) (agent v18))
  (p30 (object1 v18) (object2 v17) (rel (m16 (lex used to ))))
  (p29 (object v17) (property (m15 (lex broad))))
  (p28 (class (m8 (lex view))) (member v17)))
 (cq (p37 (object v18) (property (m19 (lex depressed))))))

(m20!)

 CPU time : 0.13

*

; Add instances for the items in the above rule
(add member #view class (build lex "view"))

(m21!)

 CPU time : 0.01

* (add object *view property (build lex "broad"))

(m22!)

 CPU time : 0.01

* (add agent #fred act (build action (build lex "live") object #room))

(m24!)

 CPU time : 0.01

* (add object1 *fred rel (build lex "used-to") object2 *view)

(m26!)

 CPU time : 0.01

* (add member *room class (build lex "room"))

(m32! m30! m29! m27!)

 CPU time : 0.09

* (add possessor *room object #kolper rel (build lex "kolper"))

(m33!)

 CPU time : 0.02

* (add object1 *kolper rel (build lex "front") object2 #courtyard)
```

```
(m34!)

 CPU time : 0.01

* (add member *courtyard class (build lex "courtyard"))

(m38! m36! m35!)

 CPU time : 0.03

* (add object (build lex "kolper") property (build lex "unknown"))

(m44! m43! m42! m41! m40! m39! m38! m35! m33! m32! m30! m29! m27! m21!)

 CPU time : 0.25

*
; Ask Cassie what "kolper" means:
^(
--> defineNoun "kolper")
 Definition of kolper:
 Class Inclusions: window,
 Possible Properties: front courtyard, provides view, provides view,
 Possessive: room,
nil



 CPU time : 0.15

*


End of /home/csgrad/krishna2/cva/kolper.demo.1 demonstration.


 CPU time : 1.01

*
```

Appendix C: Demo file for Demo 1

```
;
===========================================================================
; FILENAME: kolper.demo.1
; DATE:            10/29/2004 - 11/23/2004
; PROGRAMMER:      Rahul Krishna

;; this template version:     template.demo.2003.11.17.txt

; Lines beginning with a semi-colon are comments.
; Lines beginning with "^" are Lisp commands.
; All other lines are SNePS commands.
;
; To use this file: run SNePS; at the SNePS prompt (*), type:
;
;     (demo "kolper.demo" :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)

; Load the appropriate definition algorithm:
^(load "/projects/rapaport/CVA/STN2/defun_noun.cl")

; Clear the SNePS network:
(resetnet t)

; OPTIONAL:
; UNCOMMENT THE FOLLOWING CODE TO TURN FULL FORWARD INFERENCING ON:
;
; ;enter the "snip" package:
 ^(in-package snip)
;
; ;turn on full forward inferencing:
 ^(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:
(intext "/projects/rapaport/CVA/STN2/demos/rels")

; load all pre-defined path definitions:
```

```
(intext "/projects/rapaport/CVA/mkb3.CVA/paths/paths")

; Define some other relations
(define skolem-function arg1)

; BACKGROUND KNOWLEDGE:
; =====================

; A courtyard is outside of a room.
(describe (assert forall ($cyard $room)
      &ant (build member *cyard class (build lex "courtyard"))
      &ant (build member *room class (build lex "room"))
      cq (build object1 *cyard rel (build lex "outside") object2
*room)))

; A room has windows
(describe (assert forall *room
      ant (build member *room class (build lex "room"))
      cq (build member (build skolem-function window\ of arg1 *room)
class (build lex "window"))
      cq (build object #win rel (build skolem-function window\ of arg1
*room) possessor *room)))

; A windows provides a view
(describe (assert forall $window
      ant (build member *window class (build lex "window"))
      cq (build object1 *window rel (build lex "provides") object2
(build skolem-function view\ of arg1 *window))
      cq (build member (build skolem-function view\ of arg1 *window)
class (build lex "view"))))

;if an object fronts something that is outside a room and object is
possessed by the room then it provides a view
(describe (assert forall ($x $y $k $room)
      &ant (build object1 *x rel (build lex "front") object2 *y)
      &ant (build object1 *y rel (build lex "outside") object2 *room)
      &ant (build member *room class (build lex "room"))
      &ant (build possessor *room object *x rel *k)
      cq (build object1 *x rel (build lex "provides") object2 (build
skolem-function view\ of arg1 *x))
      cq (build member (build skolem-function view\ of arg1 *x) class
(build lex "view"))))

; if x is y's z then x is a member of class z
(describe (assert forall ($x $y $z)
      ant (build object *x possessor *y rel *z)
      cq (build member *x class *z)))

;if x and y both provide views, and x & y are members of classes a & b
repectively, and class a is unknown, then class a is a subclass of
class b
(describe (assert forall ($a $b $w $x $y $z)
      &ant (build object1 *x rel (build lex "provides") object2 *w)
      &ant (build object1 *y rel (build lex "provides") object2 *z)
      &ant (build member *x class *a)
      &ant (build member *y class *b)
      &ant (build member *w class (build lex "view"))
```

```
      &ant (build member *z class (build lex "view"))
      &ant (build object *a property (build lex "unknown"))
      cq (build subclass *a superclass *b)))


; CASSIE READS THE PASSAGE:
; =========================

; The Sentence:
; When you are used to a broad view, it becomes quite depressing
; when you come to live in a room with one or two kolpers
; fronting a courtyard.

(describe (add forall ($view $agent $room $kolper $courtyard)
      &ant (build member *view class (build lex "view"))
      &ant (build object *view property (build lex "broad"))
      &ant (build object1 *agent rel (build lex "used to ") object2
*view)
      &ant (build agent *agent act (build action (build lex "live")
object *room))
      &ant (build member *room class (build lex room))
      &ant (build possessor *room object *kolper rel (build lex
"kolper"))
      &ant (build object1 *kolper rel (build lex "front") object2
*courtyard)
      &ant (build member *courtyard class (build lex "courtyard"))
      cq (build object *agent property (build lex "depressed"))))


; Add instances for the items in the above rule
(add member #view class (build lex "view"))
(add object *view property (build lex "broad"))
(add agent #fred act (build action (build lex "live") object #room))
(add object1 *fred rel (build lex "used-to") object2 *view)
(add member *room class (build lex "room"))
(add possessor *room object #kolper rel (build lex "kolper"))
(add object1 *kolper rel (build lex "front") object2 #courtyard)
(add member *courtyard class (build lex "courtyard"))
(add object (build lex "kolper") property (build lex "unknown"))

; Ask Cassie what "kolper" means:
^(defineNoun "kolper")
```

Appendix D: Run of Demo 2

```
* ;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
; FILENAME: kolper.demo.2
; DATE:           11/15/2004 - 12/1/2004
; PROGRAMMER:     Rahul Krishna

;; this template version:     template.demo.2003.11.17.txt

; Lines beginning with a semi-colon are comments.
; Lines beginning with "^" are Lisp commands.
; All other lines are SNePS commands.
;
; To use this file: run SNePS; at the SNePS prompt (*), type:
;
;     (demo "brkolper.demo" :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

*
; Load the appropriate definition algorithm:
^(
--> load "/projects/rapaport/CVA/STN2/defun_noun.cl")
; Loading /projects/rapaport/CVA/STN2/defun_noun.cl
t



 CPU time : 0.18

*
; Clear the SNePS network:
(resetnet t)

Net reset


 CPU time : 0.01

*
; OPTIONAL:
; UNCOMMENT THE FOLLOWING CODE TO TURN FULL FORWARD INFERENCING ON:
```

```
;
; ;enter the "snip" package:
; ^(in-package snip)
;
; ;turn on full forward inferencing:
; ^(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:
(intext "/projects/rapaport/CVA/STN2/demos/rels")
File /projects/rapaport/CVA/STN2/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 : 0.02

*

End of file /projects/rapaport/CVA/STN2/demos/rels


 CPU time : 0.02

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

*
sub1 implied by the path (compose object1- superclass- ! subclass
superclass-
                          ! subclass)
sub1- implied by the path (compose subclass- ! superclass subclass- !
                           superclass object1)


 CPU time : 0.00

*
super1 implied by the path (compose superclass subclass- ! superclass
object1-
                            ! object2)
super1- implied by the path (compose object2- ! object1 superclass- !
subclass
                             superclass-)


 CPU time : 0.01

*
superclass implied by the path (or superclass super1)
superclass- implied by the path (or superclass- super1-)


 CPU time : 0.00

*

End of file /projects/rapaport/CVA/mkb3.CVA/paths/paths


 CPU time : 0.01

*
; Define other relations
(define skolem-function arg1 mod head)

(skolem-function arg1 mod head)

 CPU time : 0.00

*
; BACKGROUND KNOWLEDGE:
```

```
; ====================

; A courtyard is outside of a room.
(describe (assert forall ($cyard $room)
      &ant (build member *cyard class (build lex "courtyard"))
      &ant (build member *room class (build lex "room"))
      cq (build object1 *cyard rel (build lex "outside") object2
*room)))

(m4! (forall v2 v1)
 (&ant (p2 (class (m2 (lex room))) (member v2))
  (p1 (class (m1 (lex courtyard))) (member v1)))
 (cq (p3 (object1 v1) (object2 v2) (rel (m3 (lex outside))))))

(m4!)

 CPU time : 0.01

*
; A room has windows
(describe (assert forall *room
      ant (build member *room class (build lex "room"))
      cq (build member (build skolem-function window\ of arg1 *room)
class (build lex "window"))
      cq (build object #win rel (build skolem-function window\ of arg1
*room) possessor *room)))

(m6! (forall v2) (ant (p2 (class (m2 (lex room))) (member v2)))
 (cq
  (p6 (object b1) (possessor v2)
   (rel (p4 (arg1 v2) (skolem-function window of))))
  (p5 (class (m5 (lex window))) (member (p4)))))

(m6!)

 CPU time : 0.00

*
; A window possibly provides a view
(describe (assert forall $window
      ant (build member *window class (build lex "window"))
      cq (build mod (build lex "possibly") head (build object1 *window
rel (build lex "provides") object2 (build skolem-function view\ of arg1
*window)))
      cq (build member (build skolem-function view\ of arg1 *window)
class (build lex "view"))))

(m10! (forall v3) (ant (p7 (class (m5 (lex window))) (member v3)))
 (cq
  (p11 (class (m9 (lex view)))
   (member (p8 (arg1 v3) (skolem-function view of))))
  (p10 (head (p9 (object1 v3) (object2 (p8)) (rel (m8 (lex
provides)))))
   (mod (m7 (lex possibly))))))

(m10!)
```

```
 CPU time : 0.01

*
; A window possibly does not provide a view
(describe (assert forall $window
      ant (build member *window class (build lex "window"))
      cq (build mod (build lex "possibly") head (build min 0 max 0 arg
(build object1 *window rel (build lex "provides") object2 (build
skolem-function view\ of arg1 *window))))
      cq (build member (build skolem-function view\ of arg1 *window)
class (build lex "view"))))

(m11! (forall v4) (ant (p12 (class (m5 (lex window))) (member v4)))
 (cq
  (p17 (class (m9 (lex view)))
   (member (p13 (arg1 v4) (skolem-function view of))))
  (p16
   (head
    (p15 (min 0) (max 0)
     (arg (p14 (object1 v4) (object2 (p13)) (rel (m8 (lex
provides)))))))
   (mod (m7 (lex possibly))))))

(m11!)

 CPU time : 0.00

*
;if an object fronts something that is outside a room and object is
possessed by the room then it possibly provides a view
(describe (assert forall ($x $y $k $room)
      &ant (build object1 *x rel (build lex "front") object2 *y)
      &ant (build object1 *y rel (build lex "outside") object2 *room)
      &ant (build member *room class (build lex "room"))
      &ant (build possessor *room object *x rel *k)
      cq (build mod (build lex "possibly") head (build object1 *x rel
(build lex "provides") object2 (build skolem-function view\ of arg1
*x)))
      cq (build member (build skolem-function view\ of arg1 *x) class
(build lex "view"))))

(m13! (forall v8 v7 v6 v5)
 (&ant (p21 (object v5) (possessor v8) (rel v7))
  (p20 (class (m2 (lex room))) (member v8))
  (p19 (object1 v6) (object2 v8) (rel (m3 (lex outside))))
  (p18 (object1 v5) (object2 v6) (rel (m12 (lex front)))))
 (cq
  (p25 (class (m9 (lex view)))
   (member (p22 (arg1 v5) (skolem-function view of))))
  (p24 (head (p23 (object1 v5) (object2 (p22)) (rel (m8 (lex
provides)))))
   (mod (m7 (lex possibly))))))

(m13!)

 CPU time : 0.00
```

```
*
;if an object fronts something that is outside a room and object is
possessed by the room then it possibly  does not provide a view
(describe (assert forall ($x $y $k $room)
      &ant (build object1 *x rel (build lex "front") object2 *y)
      &ant (build object1 *y rel (build lex "outside") object2 *room)
      &ant (build member *room class (build lex "room"))
      &ant (build possessor *room object *x rel *k)
      cq (build mod (build lex "possibly") head (build min 0 max 0 arg
(build object1 *x rel (build lex "provides") object2 (build skolem-
function view\ of arg1 *x))))
      cq (build member (build skolem-function view\ of arg1 *x) class
(build lex "view"))))

(m14! (forall v12 v11 v10 v9)
 (&ant (p29 (object v9) (possessor v12) (rel v11))
  (p28 (class (m2 (lex room))) (member v12))
  (p27 (object1 v10) (object2 v12) (rel (m3 (lex outside))))
  (p26 (object1 v9) (object2 v10) (rel (m12 (lex front)))))
 (cq
  (p34 (class (m9 (lex view)))
   (member (p30 (arg1 v9) (skolem-function view of))))
  (p33
   (head
    (p32 (min 0) (max 0)
     (arg (p31 (object1 v9) (object2 (p30)) (rel (m8 (lex
provides)))))))
   (mod (m7 (lex possibly))))))

(m14!)

 CPU time : 0.01

*

; if x is y's z then x is a member of class z
(describe (assert forall ($x $y $z)
      ant (build object *x possessor *y rel *z)
      cq (build member *x class *z)))

(m15! (forall v15 v14 v13) (ant (p35 (object v13) (possessor v14) (rel
v15)))
 (cq (p36 (class v15) (member v13))))

(m15!)

 CPU time : 0.00

*

;if x and y both possibly provide views, and x & y are members of
classes a & b repectively, and class a is unknown, then class a is a
subclass of class b
(describe (assert forall ($a $b $w $x $y $z)
      &ant (build mod (build lex "possibly") head (build object1 *x rel
(build lex "provides") object2 *w))
```

```
      &ant (build mod (build lex "possibly") head (build object1 *y rel
(build lex "provides") object2 *z))
      &ant (build member *x class *a)
      &ant (build member *y class *b)
      &ant (build member *w class (build lex "view"))
      &ant (build member *z class (build lex "view"))
      &ant (build object *a property (build lex "unknown"))
      cq (build subclass *a superclass *b)))

(m17! (forall v21 v20 v19 v18 v17 v16)
 (&ant (p45 (object v16) (property (m16 (lex unknown))))
  (p44 (class (m9 (lex view))) (member v21)) (p43 (class (m9)) (member
v18))
  (p42 (class v17) (member v20)) (p41 (class v16) (member v19))
  (p40 (head (p39 (object1 v20) (object2 v21) (rel (m8 (lex
provides)))))
   (mod (m7 (lex possibly))))
  (p38 (head (p37 (object1 v19) (object2 v18) (rel (m8)))) (mod (m7))))
 (cq (p46 (subclass v16) (superclass v17))))

(m17!)

 CPU time : 0.06

*

;if x is used to a view and x is depressed and a view is possibly
provided or not provided by some z, then y is not provided by that z
(describe (add forall ($x $v1 $v2 $z $view)
      &ant (build object *x property (build lex "depressed"))
      &ant (build object1 *x rel (build lex "used-to") object2 *v1)
      &ant (build member *v1 class *view)
      &ant (build mod (build lex "possibly") head (build object1 *z rel
(build lex "provides") object2 *v2))
      &ant (build mod (build lex "possibly") head (build min 0 max 0
arg (build object1 *z rel (build lex "provides") object2 *v2)))
      &ant (build member *v2 class *view)
      cq (build min 0 max 0 arg (build object1 *z rel (build lex
"provides") object2 *v2))))

(m20! (forall v26 v25 v24 v23 v22)
 (&ant (p54 (class v26) (member v24))
  (p53
   (head
    (p52 (min 0) (max 0)
     (arg (p50 (object1 v25) (object2 v24) (rel (m8 (lex
provides)))))))
   (mod (m7 (lex possibly))))
  (p51 (head (p50)) (mod (m7))) (p49 (class v26) (member v23))
  (p48 (object1 v22) (object2 v23) (rel (m19 (lex used-to))))
  (p47 (object v22) (property (m18 (lex depressed)))))
 (cq (p52)))

(m20!)

 CPU time : 0.18
```

```
*


; if not(x provides y), then x "does not provide" y
(describe (add forall ($x $y)
      ant (build min 0 max 0 arg (build object1 *x rel (build lex
"provides") object2 *y))
      cq (build object1 *x rel (build lex "does not provide") object2
*y)))

(m22! (forall v28 v27)
 (ant
  (p80 (min 0) (max 0)
   (arg (p79 (object1 v27) (object2 v28) (rel (m8 (lex provides)))))))
 (cq (p81 (object1 v27) (object2 v28) (rel (m21 (lex does not
provide))))))

(m22!)

 CPU time : 0.04

*


; CASSIE READS THE PASSAGE:
; =========================

; The Sentence:
; When you are used to a broad view, it becomes quite depressing
; when you come to live in a room with one or two kolpers
; fronting a courtyard.

(describe (add forall ($view $agent $room $kolper $courtyard)
      &ant (build member *view class (build lex "view"))
      &ant (build object *view property (build lex "broad"))
      &ant (build object1 *agent rel (build lex "used-to") object2
*view)
      &ant (build agent *agent act (build action (build lex "live")
object *room))
      &ant (build member *room class (build lex room))
      &ant (build possessor *room object *kolper rel (build lex
"kolper"))
      &ant (build object1 *kolper rel (build lex "front") object2
*courtyard)
      &ant (build member *courtyard class (build lex "courtyard"))
      cq (build object *agent property (build lex "depressed"))))

(m26! (forall v33 v32 v31 v30 v29)
 (&ant (p90 (class (m1 (lex courtyard))) (member v33))
  (p89 (object1 v32) (object2 v33) (rel (m12 (lex front))))
  (p88 (object v32) (possessor v31) (rel (m25 (lex kolper))))
  (p87 (class (m2 (lex room))) (member v31))
  (p86 (act (p85 (action (m24 (lex live))) (object v31))) (agent v30))
  (p84 (object1 v30) (object2 v29) (rel (m19 (lex used-to))))
  (p83 (object v29) (property (m23 (lex broad))))
  (p82 (class (m9 (lex view))) (member v29)))
 (cq (p91 (object v30) (property (m18 (lex depressed))))))
```

```
(m26!)

 CPU time : 0.17

*
; Add instances for the items in the above rule
(add member #view class (build lex "view"))

(m27!)

 CPU time : 0.04

* (add object *view property (build lex "broad"))

(m28!)

 CPU time : 0.02

* (add agent #fred act (build action (build lex "live") object #room))

(m30!)

 CPU time : 0.03

* (add object1 *fred rel (build lex "used-to") object2 *view)

(m31!)

 CPU time : 0.03

* (add member *room class (build lex "room"))

(m42! m41! m39! m38! m35! m34! m32!)

 CPU time : 0.08

* (add possessor *room object #kolper rel (build lex "kolper"))

(m44! m43!)

 CPU time : 0.04

* (add object1 *kolper rel (build lex "front") object2 #courtyard)

(m45!)

 CPU time : 0.04

* (add member *courtyard class (build lex "courtyard"))

(m56! m55! m54! m53! m52! m51! m48! m47! m46! m40!)

 CPU time : 0.16

* (add object (build lex "kolper") property (build lex "unknown"))
```

```
(m58! m57! m52! m51! m46! m44! m43! m42! m39! m38! m34! m32! m27!)

 CPU time : 0.55

* (add object *fred property (build lex "depressed"))

(m48!)

 CPU time : 0.21

*

; Ask Cassie what "kolper" means:
^(
--> defineNoun "kolper")
 Definition of kolper:
 Class Inclusions: window,
 Possible Properties: does not provide view, does not provide view,
front courtyard,
 Possessive: room,
nil



 CPU time : 0.13

*


End of /home/csgrad/krishna2/cva/kolper.demo.2 demonstration.


 CPU time : 2.09

*
```

Appendix E: Demo file for Demo 2

```
;
=======================================================================
; FILENAME: kolper.demo.2
; DATE:           11/15/2004 - 12/1/2004
; PROGRAMMER:     Rahul Krishna

;; this template version:     template.demo.2003.11.17.txt

; Lines beginning with a semi-colon are comments.
; Lines beginning with "^" are Lisp commands.
; All other lines are SNePS commands.
;
; To use this file: run SNePS; at the SNePS prompt (*), type:
;
;     (demo "brkolper.demo" :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)

; Load the appropriate definition algorithm:
^(load "/projects/rapaport/CVA/STN2/defun_noun.cl")

; Clear the SNePS network:
(resetnet t)

; OPTIONAL:
; UNCOMMENT THE FOLLOWING CODE TO TURN FULL FORWARD INFERENCING ON:
;
; ;enter the "snip" package:
; ^(in-package snip)
;
; ;turn on full forward inferencing:
; ^(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:
(intext "/projects/rapaport/CVA/STN2/demos/rels")

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

```
; Define other relations
(define skolem-function arg1 mod head)

; BACKGROUND KNOWLEDGE:
; =====================

; A courtyard is outside of a room.
(describe (assert forall ($cyard $room)
      &ant (build member *cyard class (build lex "courtyard"))
      &ant (build member *room class (build lex "room"))
      cq (build object1 *cyard rel (build lex "outside") object2
*room)))

; A room has windows
(describe (assert forall *room
      ant (build member *room class (build lex "room"))
      cq (build member (build skolem-function window\ of arg1 *room)
class (build lex "window"))
      cq (build object #win rel (build skolem-function window\ of arg1
*room) possessor *room)))

; A window possibly provides a view
(describe (assert forall $window
      ant (build member *window class (build lex "window"))
      cq (build mod (build lex "possibly") head (build object1 *window
rel (build lex "provides") object2 (build skolem-function view\ of arg1
*window)))
      cq (build member (build skolem-function view\ of arg1 *window)
class (build lex "view"))))

; A window possibly does not provide a view
(describe (assert forall $window
      ant (build member *window class (build lex "window"))
      cq (build mod (build lex "possibly") head (build min 0 max 0 arg
(build object1 *window rel (build lex "provides") object2 (build
skolem-function view\ of arg1 *window))))
      cq (build member (build skolem-function view\ of arg1 *window)
class (build lex "view"))))

;if an object fronts something that is outside a room and object is
possessed by the room then it possibly provides a view
(describe (assert forall ($x $y $k $room)
      &ant (build object1 *x rel (build lex "front") object2 *y)
      &ant (build object1 *y rel (build lex "outside") object2 *room)
      &ant (build member *room class (build lex "room"))
      &ant (build possessor *room object *x rel *k)
      cq (build mod (build lex "possibly") head (build object1 *x rel
(build lex "provides") object2 (build skolem-function view\ of arg1
*x)))
      cq (build member (build skolem-function view\ of arg1 *x) class
(build lex "view"))))

;if an object fronts something that is outside a room and object is
possessed by the room then it possibly  does not provide a view
(describe (assert forall ($x $y $k $room)
      &ant (build object1 *x rel (build lex "front") object2 *y)
```

```
      &ant (build object1 *y rel (build lex "outside") object2 *room)
      &ant (build member *room class (build lex "room"))
      &ant (build possessor *room object *x rel *k)
      cq (build mod (build lex "possibly") head (build min 0 max 0 arg
(build object1 *x rel (build lex "provides") object2 (build skolem-
function view\ of arg1 *x))))
      cq (build member (build skolem-function view\ of arg1 *x) class
(build lex "view"))))


; if x is y's z then x is a member of class z
(describe (assert forall ($x $y $z)
      ant (build object *x possessor *y rel *z)
      cq (build member *x class *z)))


;if x and y both possibly provide views, and x & y are members of
classes a & b repectively, and class a is unknown, then class a is a
subclass of class b
(describe (assert forall ($a $b $w $x $y $z)
      &ant (build mod (build lex "possibly") head (build object1 *x rel
(build lex "provides") object2 *w))
      &ant (build mod (build lex "possibly") head (build object1 *y rel
(build lex "provides") object2 *z))
      &ant (build member *x class *a)
      &ant (build member *y class *b)
      &ant (build member *w class (build lex "view"))
      &ant (build member *z class (build lex "view"))
      &ant (build object *a property (build lex "unknown"))
      cq (build subclass *a superclass *b)))


;if x is used to a view and x is depressed and a view is possibly
provided or not provided by some z, then y is not provided by that z
(describe (add forall ($x $v1 $v2 $z $view)
      &ant (build object *x property (build lex "depressed"))
      &ant (build object1 *x rel (build lex "used-to") object2 *v1)
      &ant (build member *v1 class *view)
      &ant (build mod (build lex "possibly") head (build object1 *z rel
(build lex "provides") object2 *v2))
      &ant (build mod (build lex "possibly") head (build min 0 max 0
arg (build object1 *z rel (build lex "provides") object2 *v2)))
      &ant (build member *v2 class *view)
      cq (build min 0 max 0 arg (build object1 *z rel (build lex
"provides") object2 *v2))))



; if not(x provides y), then x "does not provide" y
(describe (add forall ($x $y)
      ant (build min 0 max 0 arg (build object1 *x rel (build lex
"provides") object2 *y))
      cq (build object1 *x rel (build lex "does not provide") object2
*y)))
```

```
; CASSIE READS THE PASSAGE:
; ========================

; The Sentence:
; When you are used to a broad view, it becomes quite depressing
; when you come to live in a room with one or two kolpers
; fronting a courtyard.

(describe (add forall ($view $agent $room $kolper $courtyard)
      &ant (build member *view class (build lex "view"))
      &ant (build object *view property (build lex "broad"))
      &ant (build object1 *agent rel (build lex "used-to") object2
*view)
      &ant (build agent *agent act (build action (build lex "live")
object *room))
      &ant (build member *room class (build lex room))
      &ant (build possessor *room object *kolper rel (build lex
"kolper"))
      &ant (build object1 *kolper rel (build lex "front") object2
*courtyard)
      &ant (build member *courtyard class (build lex "courtyard"))
      cq (build object *agent property (build lex "depressed"))))

; Add instances for the items in the above rule
(add member #view class (build lex "view"))
(add object *view property (build lex "broad"))
(add agent #fred act (build action (build lex "live") object #room))
(add object1 *fred rel (build lex "used-to") object2 *view)
(add member *room class (build lex "room"))
(add possessor *room object #kolper rel (build lex "kolper"))
(add object1 *kolper rel (build lex "front") object2 #courtyard)
(add member *courtyard class (build lex "courtyard"))
(add object (build lex "kolper") property (build lex "unknown"))
(add object *fred property (build lex "depressed"))


; Ask Cassie what "kolper" means:
^(defineNoun "kolper")
```

References

1. van Daalen-Kapteijns, M.M., & Elshout-Mohr, M. (1981), "The Acquisition of Word Meanings as a Cognitive Learning Process", *Journal of Verbal Learning and Verbal Behavior* 20: 386-399.

2. Martins, João P. (2002), Section on SNePS from draft of forthcoming knowledge representation text [http://www.cse.buffalo.edu/~rapaport/663/S02/martinssneps.pdf]

3. Napieralski, Scott (2003), “Noun Algorithm Case Frames”, [http://www.cse.buffalo.edu/~rapaport/CVA/CaseFrames/case-frames/]

4. Rapaport, William J., & Ehrlich, Karen (2000), "A Computational Theory of Vocabulary Acquisition", in Lucja M. Iwanska & Stuart C. Shapiro (eds.), Natural Language Processing and Knowledge Representation: Language for Knowledge and Knowledge for Language (Menlo Park, CA/Cambridge, MA: AAAI Press/MIT Press): 347-375.

5. Shapiro, Stuart C., & Rapaport, William J. (1987), "SNePS Considered as a Fully Intensional Propositional Semantic Network", in Nick Cercone & Gordon McCalla (eds.), *The Knowledge Frontier: Essays in the Representation of Knowledge* (New York: Springer-Verlag): 262-315.

6. Winston, P. H. (1977), *Artificial Intelligence*, (Reading, Mass.: Addison-Wesley)