Computationally Defining ‘Pateroller’ via Contextual Vocabulary Acquisition

Nan Meng

Nov 24, 2008

CSE663: Advanced Knowledge Representation & Reasoning

Abstract

The contextual vocabulary acquisition (CVA) project is aimed at increasing our understanding of how readers obtain a meaning of an unknown word from context, developing algorithms to implement the theory, and then in turn, helping humans in reading comprehension. In this paper, we use the SNePS knowledge representation and reasoning system to represent the information in a text passage that contains the unfamiliar word ‘pateroller’, combined with background knowledge obtained from the “think-aloud” protocol practiced with human subjects, to compute a meaning for the word ‘pateroller’ from its context. The SNePS agent Cassie is given the representation of the passage and the background knowledge to make inference about a world containing a ‘pateroller’ object. Then we run the noun algorithm to ask Cassie what propositions she made about the unknown word. In this paper, the human ‘think-aloud’ protocol is described. The representations of the passage and the background knowledge are given. Finally, the result is shown, problems and future work are also discussed.

1. Introduction

Contextual Vocabulary Acquisition (CVA) is the active, deliberate acquisition of word meanings from text by reasoning from contextual cues, which include background knowledge, and hypotheses developed from prior encounters with the word, but without external sources of help such as dictionaries or people (Rapaport, 2005). The fact involved in reading and vocabulary acquisition is that people learn most of the words and phrases frequently by making inferences and drawing conclusions from the context, instead of being taught or seeking external sources of help such as dictionaries and other people. In other words, when people encounter an unknown word, they tend to guess a meaning for that word all by themselves in the first place by reasoning from available information they have in their mind. The information includes both the reader's "mental model" of the word's "textual context", which is the information drawn from the surrounding texts of the unknown word, and the reader's prior knowledge, which is the reader's knowledge of the world together with hypotheses developed from prior encounters with the unknown word (Rapaport, 2005). Each time the unknown word is encountered, new inferences are made by adding new information and eliminating inappropriate hypotheses.

The objectives of the CVA project include increasing our understanding of the contextual vocabulary acquisition process, using the observations to form our computational theory of CVA, developing computer programs that implement and test this theory, and developing a curriculum to improve students' abilities to use CVA. [1] The computational implementation of the CVA project is based on the SNePS knowledge representation and reasoning system,

[1] http://www.cse.buffalo.edu/~rapaport/CVA/cvadescription.html
Copyright © 2002 by William J. Rapaport (rapaport@cse.buffalo.edu)
title: Contextual Vocabulary Acquisition: description
date of last access: Nov/24/2008

which is “a programming language whose primary data structure is semantic network (a labeled directed graph), with commands for building such networks and finding nodes in such a network given arbitrary descriptions” (Shapiro and Rapaport, 1995).

In this particular project, we use SNePSUL, the SNePS user language, to represent our text passage and background knowledge. Then we give our representations as input to the SNePS computational agent named Cassie. By using the SNePSUL standard case frames, which will be described in detail shortly, we can ask Cassie to use the noun algorithm that will then give a dictionary-like definition for the unknown word ‘pateroller’. In this paper, we are mainly concerned with finding the different ways of representing information in the text passage and background knowledge in SNePS.

2. The Passage and the Unknown Word

Our passage is provided by Dr. Michael Kibby, and is taken from Morrison, Tony. (1987, 2004). Beloved. New York: Vintage. The original passage is as the following:

“On a riverbank in the cool of a summer evening two women struggle under a shower of silvery blue. They never expected to see each other again in this world and at that moment couldn’t care less. But there on a summer night surrounded by blue fern they did something together appropriately and well. A pateroller passing would have sniggered to see two throwaway people, two lawless outlaws—a slave and a barefoot white woman with unpinned hair—wrapping a ten-minute-old baby in the rags they wore. But no pateroller came and no preacher. The water sucked and swallowed itself beneath them. There was nothing to disturb them at their work. So they did it appropriately well."

The unknown word within this passage is the noun 'pateroller'. Besides the original passage from the book, additional background information is needed to form a richer context, and is provided by Dr. Michael Kibby as the following statements:

"The story takes place in the South and in Ohio in the pre- and post-Civil War days. Sethe is a runaway, female slave who is about to give birth to her fourth child. The other three children have already escaped are in southern Ohio. Sethe is, of course, very afraid, because if she is caught by the law, not only will she be returned to her master, but she will be beaten, perhaps even crippled so she could not run again.

"As Sethe is running, she meets Amy, a young, white girl also running away from her sharecropping parents' home. She is headed for Boston, in search of velvet. They befriend each other as they walk toward Ohio and, for Sethe, freedom, and for Amy, escape. The slip quietly through the nights in the forests and pastures, shunning roads to avoid being seen by anyone for fear of being caught. After several nights march, they come to the Ohio river and steal a boat to cross the river. Minutes before they reach the Ohio shore, Sethe—with Amy's help—gives birth to her baby. This scene is ten minutes later, and they have just reached the Ohio shore and tied up to a tree."

It is hard to find the definition of the noun “pateroller” in a formal dictionary, but the definition can be found on the Internet, given as follows: ‘pateroller’ is a term derived from the word ‘patrollers’, a reference to the men who patrolled the highways and byways in search of runaway slaves. [1]

3. The Human Protocol

To acquire the background knowledge needed for reasoning about the meaning of the unknown word, experiments are carried out first with human subjects, who don’t know the meaning of that word presumably, by showing them the passage and asking them to figure out a meaning for that word, and then tell all the information (i.e. background knowledge and contextual information) they used in order to get that meaning. We keep a record of everything piece of information they used. In this experiment, all the information the human subjects used to understand the unknown word is called the ‘think-aloud’ protocol. In our project, two people participate in the experiment; one of them (subject A) is a 19-year-old female college student, the other one (subject B) is a 50-year-old male lawyer who has a lot of reading experience. The following are their ‘think-aloud’ protocols.

[1] http://www.geocities.com/Heartland/Woods/3501/19th.htm
authors: Dan and Nathan Lee, Southeast Louisiana Living History Association
title: 19th Century Amusements: Games & Toys
date of last access: Nov/25/2008

Subject A: “The pateroller would have sniggered at black people and poor people. It made me think about prejudice, anti-black, and wealthy.” “The passage also mentioned that ‘no pateroller came and no preacher’, so the pateroller might be similar to a preacher, a farther, or a priest, something related to the church.” “The pateroller might be a man, because he would have sniggered at two women.”

However, after some serious thought, except the feelings listed above, subject A was not able to give a definition for our unknown word.

Subject B: “I first noticed the time period, Civil War, which was about from 1860 to 1865 and was mainly about slavery, also where the story happened, Southern Ohio. Then I started to think what people were around there. I also noticed that water is involved, so paterollers might be on ways or water.” “I noticed the word ‘sniggering’. What’s the reason of sniggering? What type of person would be sniggering? They might be high and mighty law person, since the two people in this story were ‘lawless outlaws’, so it might have something to do with the law, like they catch somebody’s slave and return, they get paid. It’s like modern time bounty hunters.” “The ‘preacher’ is mentioned, so the story might have something to do with religion whether preachers are used to make a contrast to paterollers or not is not clear.”

Finally, the subject gave the word ‘pateroller’ a meaning that he thought the most plausible: “somebody who makes money hunting down and returning runaway slaves.” We notice how close this definition is to the correct meaning of our word.

Apparently, as an experienced reader, subject B was “computing” with much more information than subject A was when they were both trying to understand the word. In addition, subject B was able to gather more information from both the passage and his own experience (background knowledge). What is also interesting is that as subject B was attracted by the occurrences of ‘water’ in the context, he was also trying to give his unknown word other definitions, such as “people on the water doing transportation”. Given the correct meaning of the word, we know that subject B was really distracted by the appearances of ‘water’ in the passage, however, what he did was a very good example of contextual vocabulary acquisition as he was using every possible piece of information in the context to figure out a meaning of the word. Moreover, subject B’s CVA process was quite successful as he was able to give a very close definition as his best bet, and meanwhile still considering other less possible definitions as candidates. It might be worth mentioning that to understand a word or to do reasoning, besides the background knowledge and contextual information, a mechanism to decide which piece of information or knowledge is more relevant and useful might also be necessary.

In our approach, not all the information from the text passage and the ‘think-aloud’ protocol are used, since much of the knowledge are not useful for figuring out the most appropriate meaning of our unknown word, so we trim the passage and background knowledge leaving only helpful information. This will be discussed in more details in the next section.

4. SNePS Representation

4.1 SNePS case frames

There are several standard case frames that can be recognized by the noun algorithm. [1] We use the following ones in our representation.

agent/act/action

agent/act/action/object

lex

member/class

object/proper-name

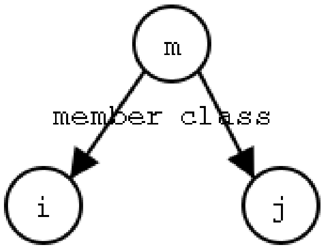
object/property

object/rel/possessor

synonym/synonym

For a simple example, here is the SNePS semantic network representation and the semantics of the following standard case frame.

member/class



Semantics: [[m]] is the proposition that [[i]] is a member of the class [[j]].

[1] http://www.cse.buffalo.edu/~rapaport/CVA/cvaresources.html
Copyright © 2002-2006 by William J. Rapaport (rapaport@cse.buffalo.edu)
title: Contextual Vocabulary Acquisition: resources
date of last access: Dec/01/2008

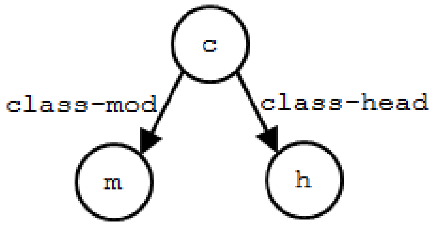
We also use the following case frames from the SNePS case frame dictionary. [1] The syntax and semantics for these case frames can be found in this dictionary.

forall/ant/cq

min/max/arg

In addition, we use the following non-standard case frame to represent composite class concepts that consist of a modifier component and a head component, i.e. slave master.

class-mod/class-head

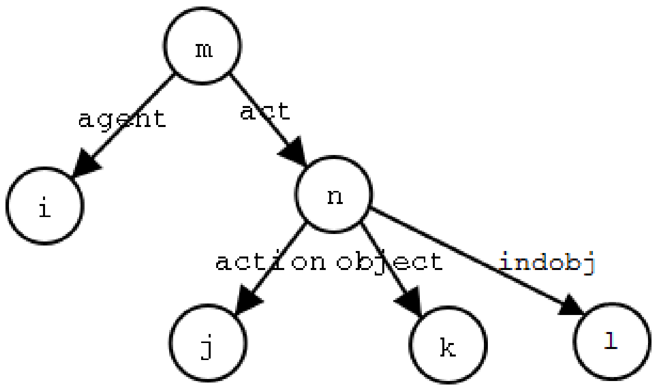


Semantics: [[c]] is the concept of the class expressed by combining the modifier concept [[m]] and the head concept [[h]].

Another non-standard case frame we use is as follows, which is to represent the indirect object information in action related propositions.

agent/act/action/object/indobj

[1] http://www.cse.buffalo.edu/sneps/Bibliography/bibliography.html#Manuals
author: Stuart C. Shapiro
title: The SNePS Research Group Bibliography
date of last access: Dec/01/2008



Semantics: [[m]] is the proposition that agent [[i]] performs action [[j]] directly to object [[k]] and indirectly to object [[l]].

4.2 Background Knowledge Representation

As we follow our human 'think-aloud' protocol, some information from the human subjects turns out to be less useful for the reasoning about our unknown word. On the other hand, some critical information is missing in the reasoning chain. We want to call the reasoning steps a 'chain' because all the information we provide to Cassie as background knowledge and passage information should be able to form a set of propositions that is then followed step by step from the primitive propositions (i.e. information told by the passage) to the expected propositions (i.e. propositions about a meaning of the unknown word). Therefore, our work is to do the knowledge engineering by selecting those background information that are most useful and relevant to our particular task, and are as general as possible.`

The complete SNePSUL representation of our background knowledge and corresponding semantic network diagrams is given in Appendix A. In the following part, we give the 'English-like' description of the background knowledge we tell Cassie, together with our explanations and discussions.

With the first five rules, we tell Cassie the general relationships among slaves, slave masters, and bounty hunters, and tell Cassie their possible actions and corresponding consequences that are useful for the reasoning about our unknown word. Notice that there are a huge number of rules or facts about the entities (i.e. slaves, bounty hunters) in the world of the passage; however, we are only telling Cassie those rules or facts that we are interested in. For example, to continue our reasoning with further rules, we are interested in whether the bounty hunters are getting rewards, so that they might get happy and snigger, but we are not interested in whether the slave masters are male or female, which is not closely related to the meaning of our unknown word.

RULE1: If *a* is a slave master, and *b* is a slave, and *b* is *a*'s slave, then *a* puts bounty on *b*.
RULE2: If *a* puts bounty on *b*, and *c* returns *b* to *a*, then *c* gets rewards (from *a*).
RULE3: If *a* is a bounty hunter, then *a* wants bounty.
RULE4: If *a* is a bounty hunter, and *b* puts bounty on *c*, and *a* sees *c*, then *a* catches *c*.
RULE5: If *a* is a bounty hunter, and *b* puts bounty on *c*, and *a* catches *c*, then *a* returns *c* to *b*.

One of the things we are trying to do in this project is figuring out a way to represent and reason about the class concepts consisting of the modifier and head components. For example, for the slave master concept, the class modifier is ‘slave’, and the class head component is ‘master’. Our approach to this problem is using the *class-mod/class-head* case frame introduced earlier, where the class concepts are expressed by combining the modifier and head components, however, how we shall combine the two components, or in other words, what kind of relationship we shall give them, still remains as a problem. Maybe there exist a fixed number of patterns between all modifier and head pairs, but that’s not the main topic of this paper, and shall be considered a future work. In previous CVA papers, people argued that under certain circumstances, instead of using a modifier-head case frame, some complex class concepts are better expressed as single entities (Goldfain, 2003).

The next three rules deal with the action of sniggering and its consequence. Notice that here we are trying to relate the action of sniggering with the action of getting benefit, so that with further rules about bounty hunters and their actions, possibly getting benefit and reward; we can indicate the identity of the bounty hunter from its action of sniggering. Since in the passage it is the pateroller who might be sniggering, with the rules we build according to the human protocol, our goal is to draw the relationship between a pateroller and a bounty hunter.

RULE6: If *a* sniggers, then *a* is being smug and mean.

RULE7: If *a* sniggers, then *a* laughs for him/herself.

RULE8: If *a* laughs for him/herself, then *a* gets some benefit.

Also notice that we are trying to make our rules as general as possible. That is why instead of combining RULE7 and RULE8 by telling Cassie a rule like “If *a* sniggers, then *a* gets some benefit.”, which is not general enough since people snigger for a lot of different reasons, we break the rules apart so that hopefully our rules hold in all situations. However, we have to admit that we cannot assert many general rules that hold in every situation, simply because there are not many of them in the real world and exceptions always exist. Therefore, our goal is to make the rules hold in most situations.

The following four rules tell Cassie how to reason about the direct and indirect objects of actions. RULE9 and RULE10 intuitively tell Cassie that a given object may be represented by many different names, so that Cassie can express the object of an action with told synonymous terminologies. With these rules Cassie will be able to know that if someone is getting bounty, then he/she is getting reward or benefit. RULE11 and RULE12 tell Cassie that if an agent applies an action to a whole class of things, then that agent will apply the action to a member of that class. These two rules help Cassie to reason about those facts such as since bounty hunters return slaves, and Sethe is a slave, then bounty hunters will also return Sethe.

RULE9: If *a* takes action *act* directly on *b*, and *b* and *c* are synonyms, then *a* takes action *act* directly on *c*.

RULE10: If *a* takes action act directly on *b* indirectly on *d*, and *b* and *c* are synonyms, then *a* takes action *act* directly on *c* indirectly on *d*.

RULE11: If *a* takes action *act* directly on *c*, and *b* is a member of class *c*, then *a* takes action *act* directly on *b*.

RULE12: If *a* takes action *act* directly on *c* indirectly on *d*, and *b* is a member of class *c*, then *a* takes action *act* directly on *b* indirectly on *d*.

The last rules tell Cassie the relationship between the action of wanting and the status of having some expectation, among other facts. Here we connect sniggering and expectation by analyzing the motivation behind sniggering, that is people snigger when they are getting something they expect. Then with RULE17, we tell Cassie how to connect the action of getting bounty and the status of being a bounty hunter. This can be viewed as the last step in our reasoning chain.

RULE13: If *a* wants *b*, and *a* gets *b*, then *a* is happy.

RULE14: If *e* is *a*'s expectation, then *e* is an expectation, and *a* wants *e*.

RULE15: If *a* wants *e*, then *e* is *a*'s expectation.

RULE16: If *a* sniggers, and *e* is *a*'s expectation, then *a* gets *e*.

RULE17: If *a* gets bounty, then *a* is a bounty hunter.

4.3 Passage Representation

There is a lot of information in our passage, but only a small portion of it is very useful to figure out a meaning of our unknown word. Thus we eliminate extra information and only represent those pieces that are crucial for our purpose. So the trimmed version of the passage is as follows:

“Sethe is a slave.”

“Sethe and Amy are outlaws.”

“A pateroller passing would have sniggered to see two outlaws.”

We then use several propositions to represent the information in each of the three sentences. Here we list our SNePSUL representation. Some of the semantic network representations are given in Appendix B.

In the representations of the first sentence, we build two nodes representing the concept of someone named Sethe and the concept of a slave master respectively. Then we tell Cassie that Sethe is the master’s slave by using the *object/rel/possessor* case frame. In this way, we fully represent all the information in the first sentence, including the existence of a slave master who owns Sethe. This information is expressed implicitly and is also useful for our reasoning.

```
;--------------------
; Sethe is a slave.
;--------------------
; There is someone named Sethe.
(describe (add object #sethe
              proper-name (build lex "Sethe"))
              = someone-is-named-sethe)
```

```
; Sethe is a slave.
(describe (add member *sethe
              class (build lex "slave"))
              = sethe-is-a-slave)

; Someone is a slave master.
(describe (add member #master
              class (build  class-mod (build lex "slave")
              class-head (build lex "master")))
              = someone-is-a-slavemaster)

; Sethe is the master's slave.
(describe (add object *sethe
              rel (build lex "slave")
              possessor *master)
              = sethe-is-masters-slave)
```

The representation of the second sentence is straightforward. We first build a node representing the concept of someone named Amy. Then tell Cassie that Sethe and Amy are outlaws.

```
;---------------------------
; Sethe and Amy are outlaws.
;---------------------------
; There is someone named Amy.
(describe (add object #amy
              proper-name (build lex "Amy")))

; Sethe and Amy are outlaws.
(describe (add member (*sethe *amy)
              class (build lex "outlaw")))
```

The representation of the third sentence needs some discussion. As before, we first build the node representing the pateroller. Then we tell Cassie that the pateroller sees Sethe and Amy, which is not really the case. The same kind of problem was also encountered by previous CVA researchers, [1] where they wanted to reason about the consequence of absorbing extreme sports into the Olympics juggernaut, however, it is not the case that extreme sports are indeed absorbed. Thus in order to reason about the consequence, they simply asserted that the antecedent holds. Therefore, here we represent our sentence using the same technique by telling Cassie that the pateroller does see Sethe and Amy. So Cassie will believe that the pateroller also sniggers, which will then trigger our following reasoning steps.

```
;--------------------------------------------------------------
; A pateroller passing would have sniggered to see two outlaws.
;--------------------------------------------------------------
; "pateroller" is an unknown word.
(describe (add object (build lex "pateroller")
               property (build lex "unknown")))

; Someone is a pateroller.
(describe (add member #pt
               class (build lex "pateroller")))

; The pateroller sees Sethe and Amy.
(describe (add agent *pt
               act (build action (build lex "see")
               object (*sethe *amy))))

; If a pateroller p sees Sethe and Amy, then p will snigger.
(describe (add forall $p
         &ant ((build member *p
                     class (build lex "pateroller"))
               (build agent *p
                     act (build action (build lex "see")
                     object (*sethe *amy))))
```

[1] http://www.cse.buffalo.edu/~rapaport/CVA/Juggernaut/CVA/
author: Matthew Watkins
title: CVA project, Juggernaut, by Matthew Watkins
date of last access: Dec/05/2008

```
          cq ((build agent *p
                     act (build action (build lex "snigger"))))))
```

5 Results

After telling Cassie the SNePSUL representation of the background knowledge and our passage, we call the noun algorithm to define the unknown word ‘pateroller’, and obtain the following results:

```
^(
--> defineNoun "pateroller")
 Definition of pateroller:
 Possible Class Inclusions: m15, bounty hunter,
 Possible Actions: see bounty, see outlaw slave, see outlaw, snigger,
laugh, catch bounty, catch outlaw slave, return bounty, return
outlaw slave, get b1, get bounty, get benefit, get reward, get outlaw
slave, want b1, want bounty, want benefit, want reward,
 Possible Properties: happy, smug, mean,
nil
```

Here the node m15 represents the complex class concept of bounty hunter, and the node b1 represents the pateroller’s expectation. We are also getting the compound class name “bounty hunter” because we tried to assert some rules that can reason about the *class-modifier* and *class-head* components, and put the two parts together to form the compound class name. The rule we used is as follows:

```
; The concept of bounty hunter, has proper-name "bounty hunter"
(describe (add object (build class-mod (build lex "bounty")
                             class-head (build lex "hunter"))
```

```
                proper-name (build lex "bounty\ hunter")))
```

However, this rule is not a good practice, since the *object/proper-name* case frame is not appropriate for the concept of bounty hunter and the term "bounty hunter".

Here we are facing some problem that previous researchers also encountered, since when we were trying to obtain additional properties of paterollers, we tell Cassie that "if *a* returns *b*, and *b* is a slave, then *a* is evil." When we tell Cassie to describe all the propositions she believes, we can find the following asserted propositions, which show that Cassie believes that the pateroller returns Sethe, Sethe is a slave, and that whoever returns a slave is evil. So it should be the case that Cassie also includes 'evil' as a possible property of pateroller after invoking the noun algorithm, but she doesn't.

```
(m178! (act (m177 (action (m7 (lex return))) (object b2[Sethe])))
(agent b6[pateroller]))

(m71! (class (m1 (lex slave))) (member b2[Sethe]))

(m35! (forall v21 v20)
 (&ant (p44 (class (m1 (lex slave))) (member v21))
  (p43 (act (p42 (action (m7 (lex return))) (object v21))) (agent
v20)))
 (cq (p45 (object v20) (property (m34 (lex evil))))))
```

Previous researchers solve this problem by re-adding the existing nodes (Xu, 2004). This can also be done by explicitly asking Cassie to find the property of pateroller, using the SNePS `find` command `(describe (find (property- object) *pt)`. Then if we call the noun algorithm again, Cassie will include 'evil' as a possible property.

6 Conclusion and Future Work

In this paper, we ask the SNePS agent Cassie to use the noun algorithm to give a dictionary-like definition for the unknown word ‘pateroller’ by telling her the necessary background knowledge and context information represented in SNePSUL case frames. Cassie is able to figure out possible actions and properties of a pateroller appropriately. However, some problems are observed during the project.

The first problem is the difficulty of making general rules in our background knowledge base. Our current approach to this problem is providing Cassie with rules that meet our purpose, which means, the rules are very much limited by individual projects, and a researcher working with word A may define some rules in the knowledge base that are contradictory to the some rules defined by another researcher working with word B. Part of the reason is because we are not able to rule out all the exceptions efficiently, and cannot precisely represent the phrases such as ‘probably’ and ‘normally’ in our system. Future work may consider integrating a technique like circumscription into the representation to deal with exceptions.

The next problem is the representation and reasoning of the indirect objects of actions. Since the current version of noun algorithm does not recognize the *agent/act/action/object/indobj* case frame, when Cassie is asked to define the unknown word, she is not able to extract information from the indirect object component of the asserted propositions. Future work may consider adding this facility to the noun algorithm. However, if the only concern is to obtain an appropriate meaning for the unknown word, we can simplify our knowledge base and information in the passage, and eliminate the indirect object components to get around the problem.

Some other problems include the issue with forward inference mentioned earlier, where Cassie ignores some propositions in her knowledge base when she gives the definition, and we have to remind her about those beliefs explicitly; and the problem of expressing complex class concepts such as “slave master” or “toy gun”. Since the last problem is still an active one, for now we might just handle it using our *class-modifier/class-head* case frame and leaving the relationship between the modifier and the head components implicit.

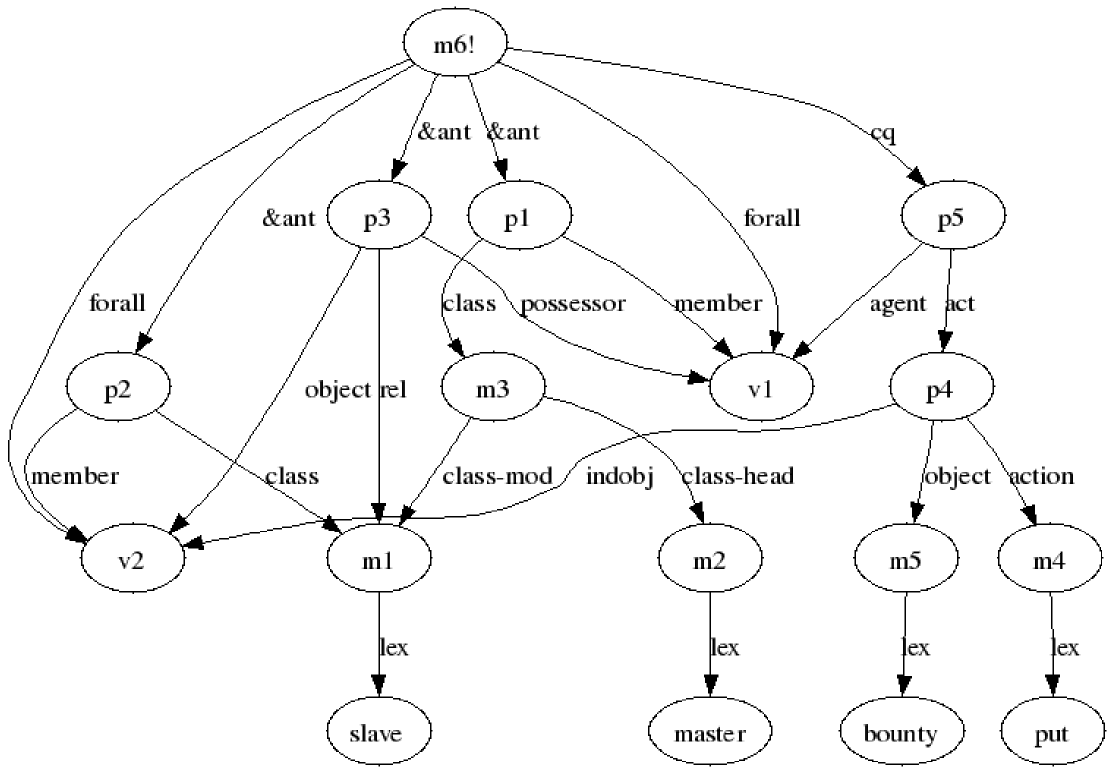
Long terms goals may include integrating the CYC knowledge base [1] into the CVA project to facilitate the background knowledge representation stage, and applying natural language processing tools on the given passages to convert the sentences to SNePSUL expressions that Cassie can process directly. However, currently these intents are still very demanding and challenging.

[1] http://www.cyc.com/cyc/opencyc/overview
author: Cycorp, Inc.
title: Overview of OpenCyc
date of last access: Dec/5/2008

Appendix A: SNePSUL representation of background knowledge

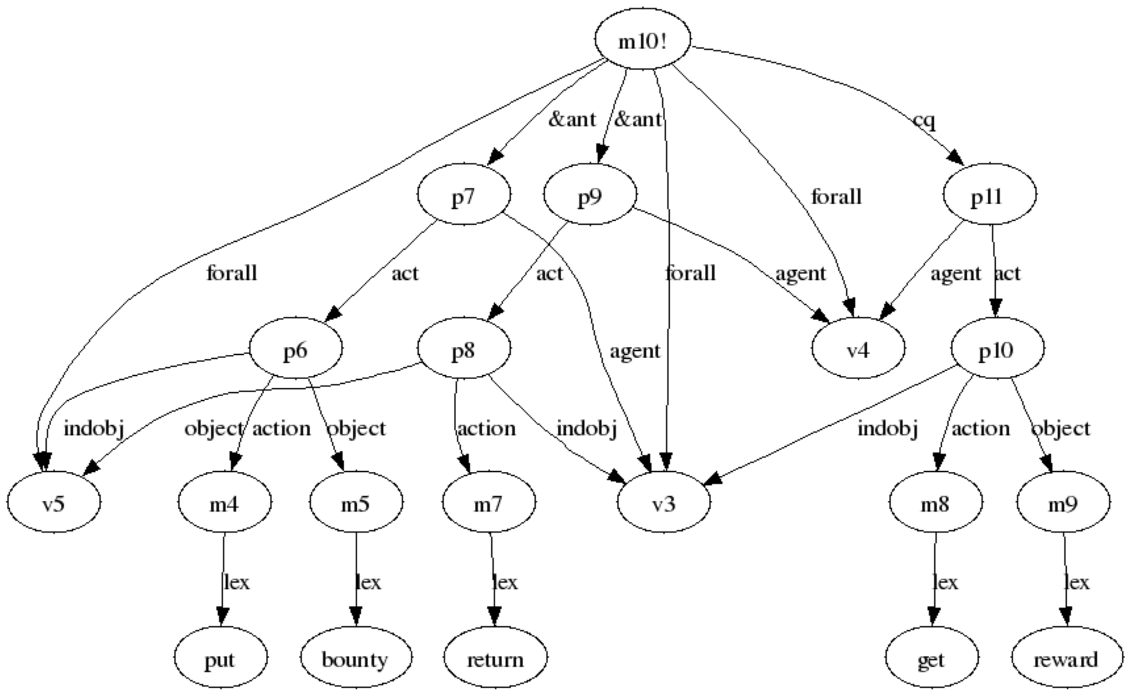
Slave masters put bounties on their slaves.

```
(describe (add forall ($sm $sl)
          &ant (    (build member *sm class (build class-mod (build lex "slave")
                                                   class-head (build lex "master")))
                    (build member *sl class (build lex "slave"))
                    (build object *sl rel (build lex "slave") possessor *sm))
          cq (      (build agent *sm
                              act (build action (build lex "put")
                              object (build lex "bounty")
                              indobj *sl)))) = master-put-bounty-on-slave)
```



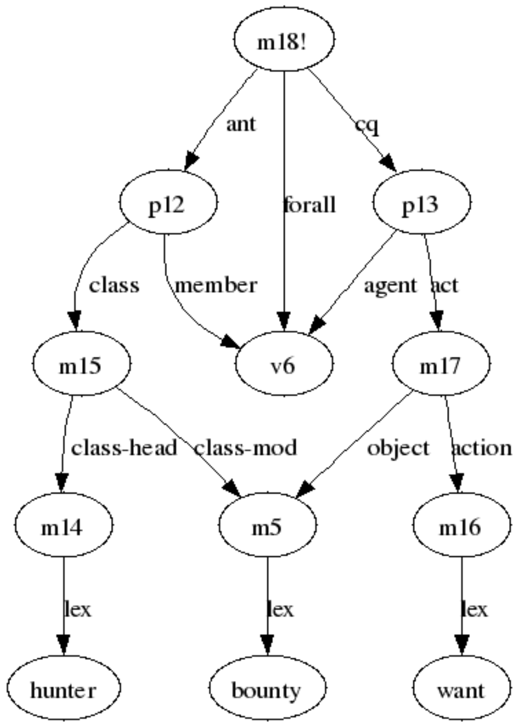
If A puts bounty on STH and B returns STH to A, then B gets rewards from A.

```
(describe (add forall ($a $b $sth)
        &ant (    (build agent *a
                        act (build action (build lex "put")
                        object (build lex "bounty")
                        indobj *sth))
                  (build agent *b
                        act (build action (build lex "return")
                        object *sth
                        indobj *a)))
        cq (      (build agent *b
                        act (build action (build lex "get")
                        object (build lex "reward")
                        indobj *a)))) = bounty-return-reward)
```



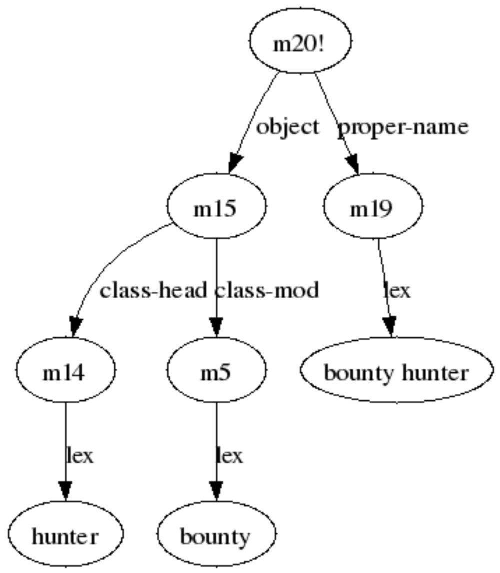
Bounty hunters want bounty.

```
(describe (add forall $b
          ant (     (build member *b class (build class-mod (build lex "bounty")
                                                  class-head (build lex "hunter"))))
          cq (      (build agent *b
                         act (build action (build lex "want")
                         object (build lex "bounty"))))) = bounty-hunter-want-bounty)
```



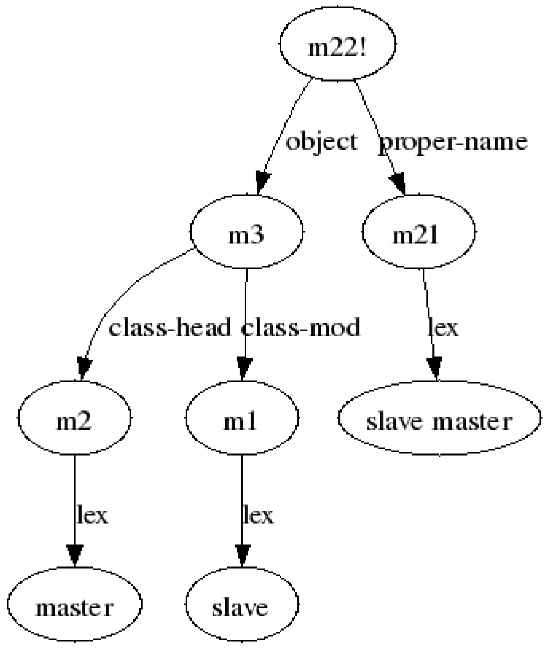
The concept of bounty hunter, has proper-name "bounty hunter"

```
(describe (add object (build class-mod (build lex "bounty")
                             class-head (build lex "hunter"))
               proper-name (build lex "bounty\ hunter")))
```



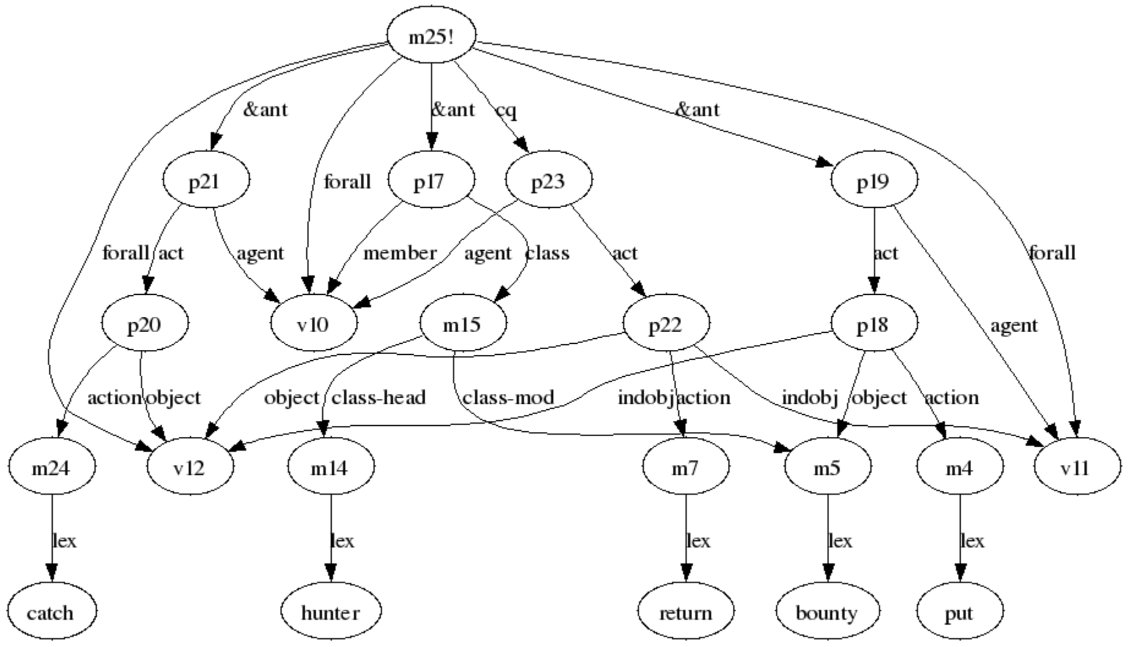
The concept of slave master, has proper-name "slave master"

```
(describe (add object (build class-mod (build lex "slave")
                             class-head (build lex "master"))
               proper-name (build lex "slave\ master")))
```



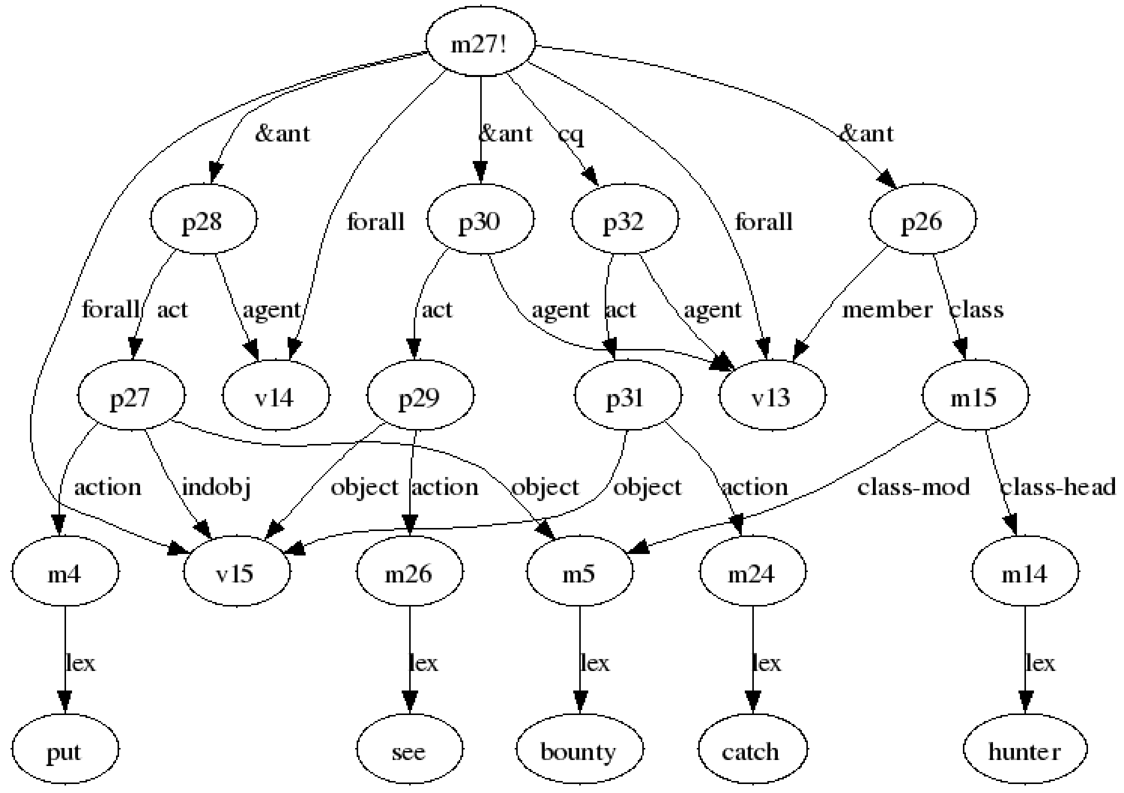
If A is a bounty hunter, and B puts bounty on C, and A catches C, then A returns C to B (and gets bounty from B).

```
(describe (add forall ($a $b $c)
            &ant (    (build member *a class (build class-mod (build lex "bounty")
                                                    class-head (build lex "hunter")))
                      (build agent *b
                          act (build action (build lex "put")
                          object (build lex "bounty")
                          indobj *c))
                      (build agent *a
                          act (build action (build lex "catch")
                          object *c)))
            cq (      (build agent *a
                          act (build action (build lex "return")
                          object *c
                          indobj *b)))))
```



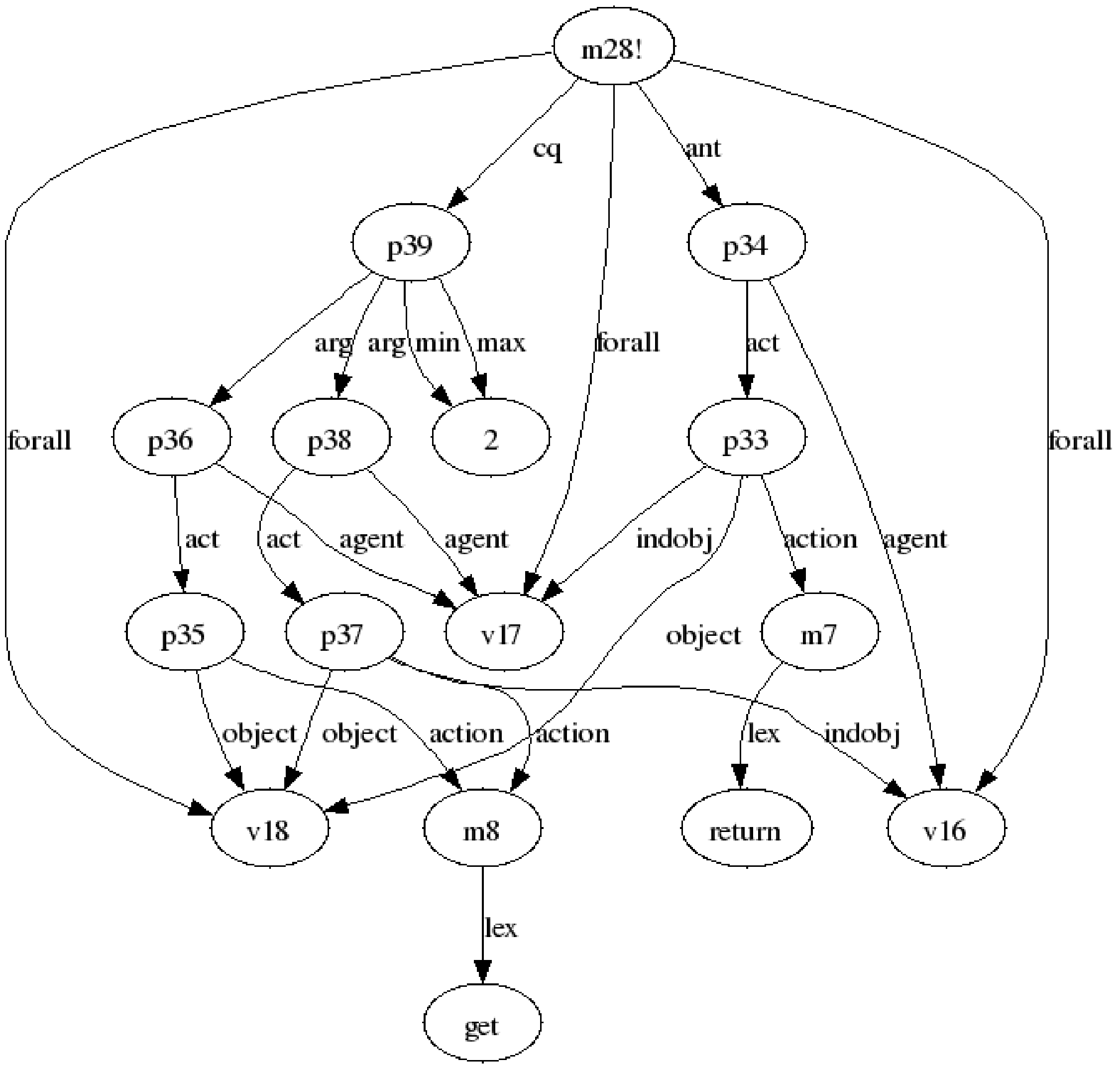
If A is a bounty hunter, and B puts bounty on C, and A sees C, then A catches C.

```
(describe (add forall ($a $b $c)
          &ant (      (build member *a class (build class-mod (build lex "bounty")
                                                   class-head (build lex "hunter")))
                      (build agent *b
                           act (build action (build lex "put")
                           object (build lex "bounty")
                           indobj *c))
                      (build agent *a
                           act (build action (build lex "see")
                           object *c)))
          cq (        (build agent *a
                           act (build action (build lex "catch")
                           object *c)))))
```



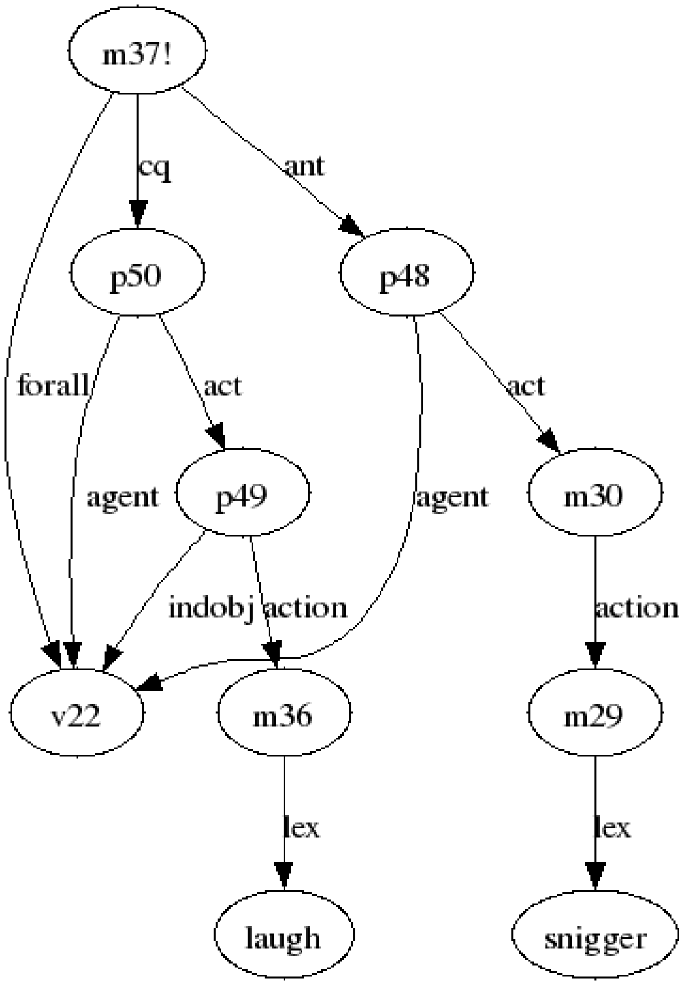
If A returns C to B, then B gets C, and B gets C from A.

```
(describe (add forall ($a $b $c)
            ant (       (build agent *a
                               act (build action (build lex "return")
                               object *c
                               indobj *b)))
            cq (        build min 2 max 2
                        arg ((build agent *b
                                     act (build action (build lex "get")
                                     object *c))
                               (build agent *b
                                     act (build action (build lex "get")
                                     object *c
                                     indobj *a))))))
```



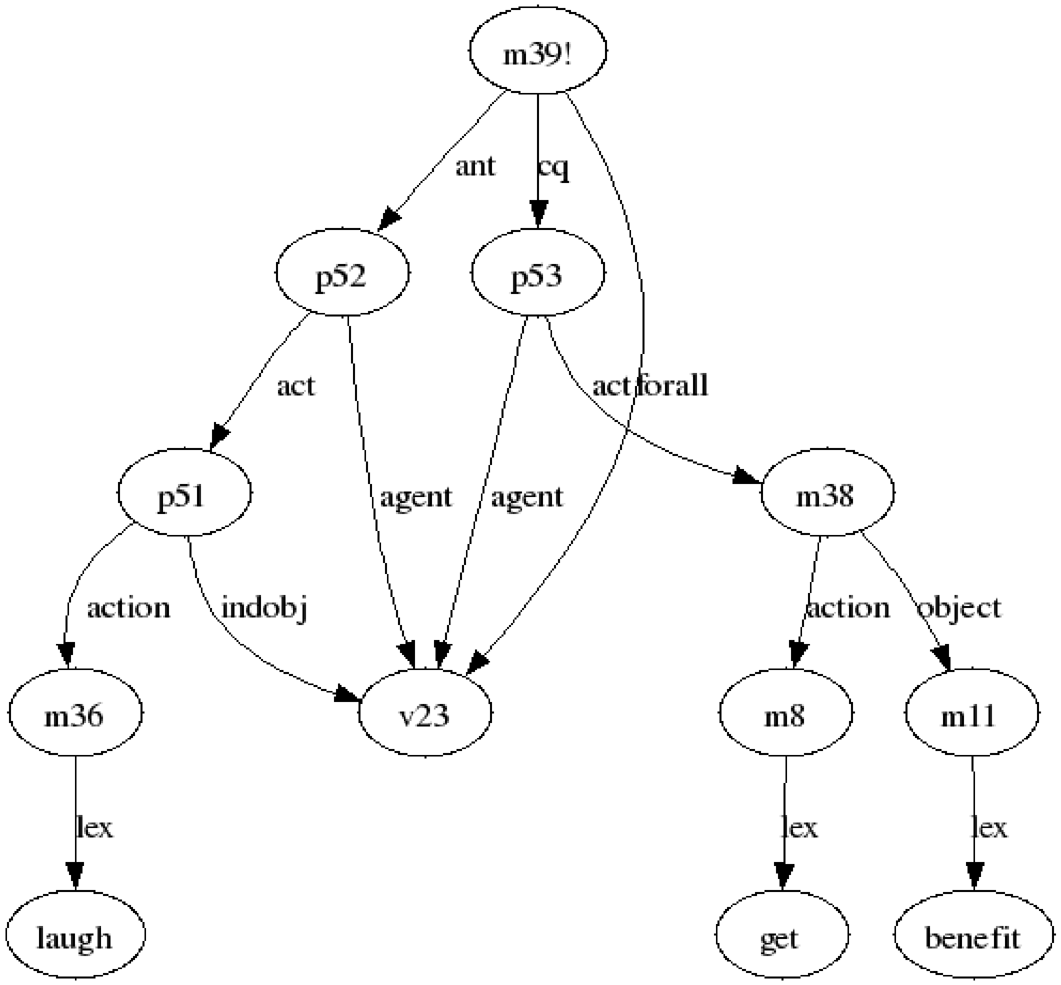
If A sniggers, then A laughs for him/herself.

```
(describe (add forall $a
          ant (     (build agent *a
                         act (build action (build lex "snigger"))))
          cq (      (build agent *a
                         act (build action (build lex "laugh")
                         indobj *a)))))
```



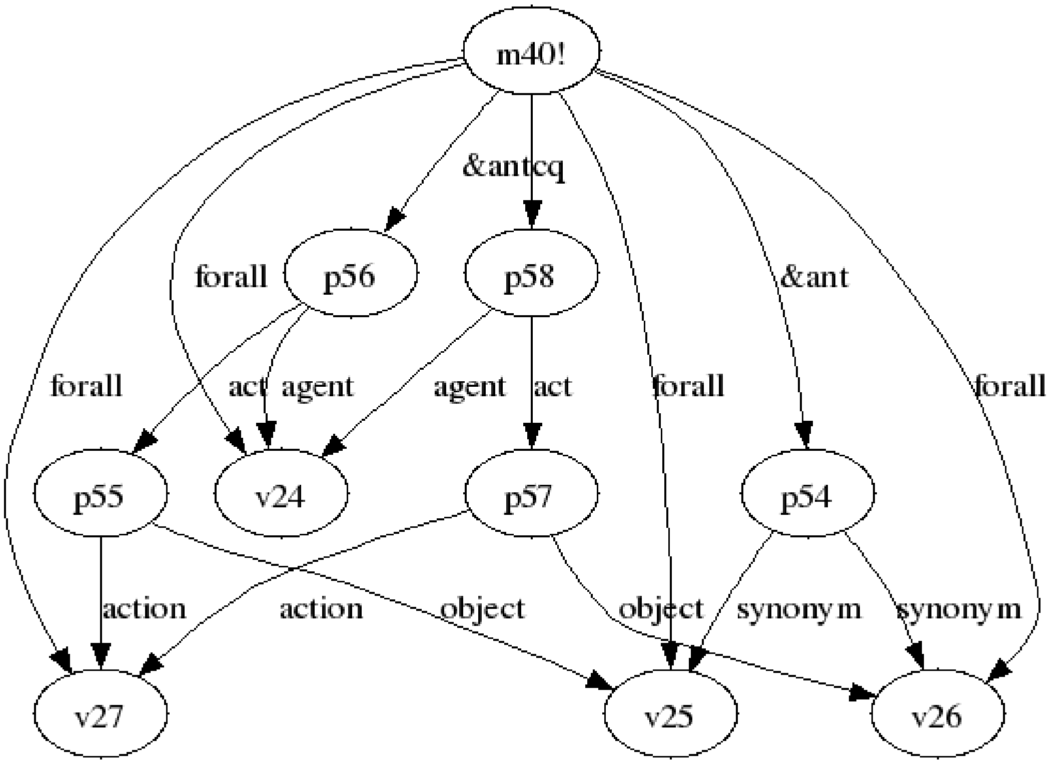
If A laughs for him/herself, then A gets some benefit.

```
(describe (add forall $a
            ant (       (build agent *a
                               act (build action (build lex "laugh")
                               indobj *a)))
            cq (         build agent *a
                               act (build action (build lex "get")
                               object (build lex "benefit")))))
```



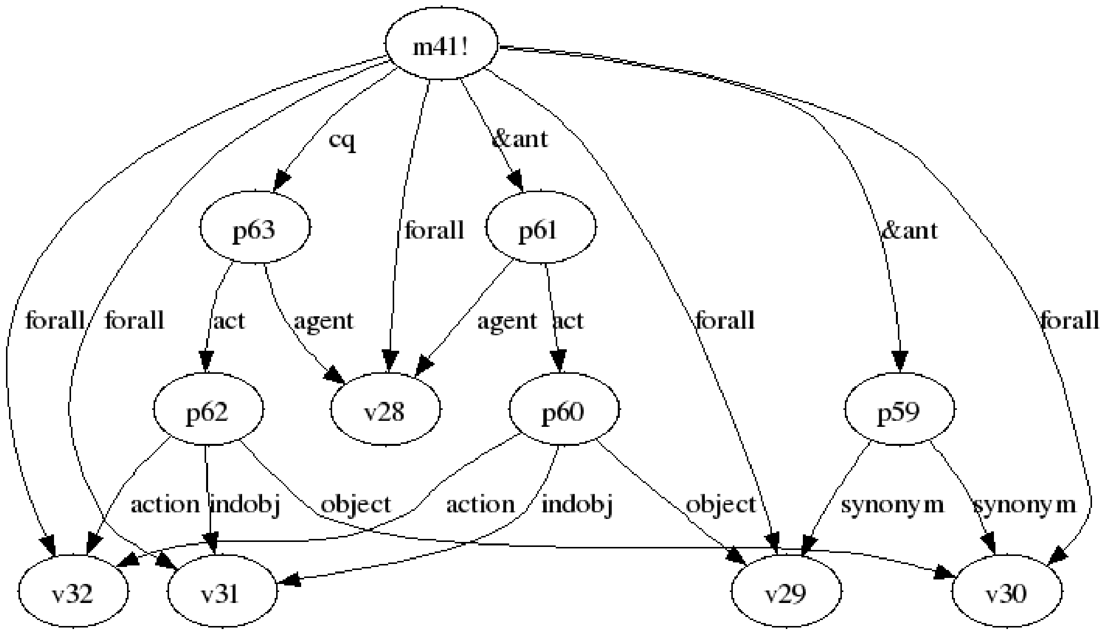
If A takes action ACT directly on B, B and C are synonyms, then A takes action ACT directly on C.

```
(describe (add forall ($a $b $c $act)
            &ant (    (build synonym *b synonym *c)
                      (build agent *a
                             act (build action *act
                             object *b)))
            cq (      build agent *a
                             act (build action *act
                             object *c))))
```



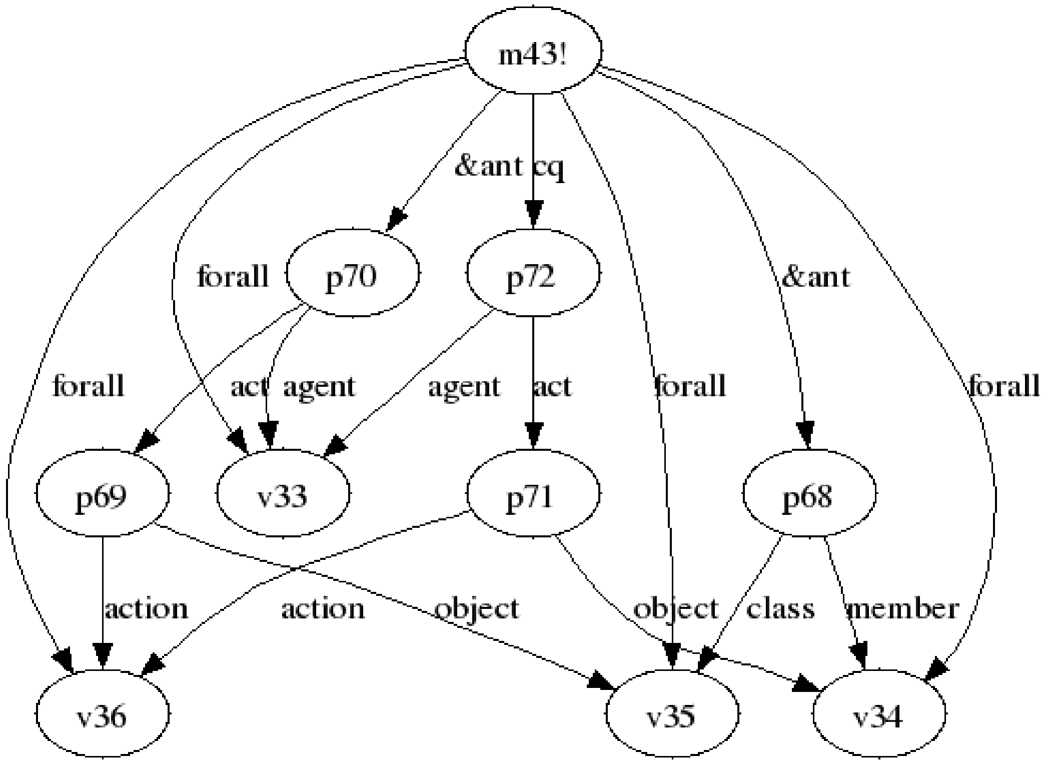
If A takes action ACT directly on B indirectly on D, B and C are synonyms, then A takes action ACT directly on C indirectly on D.

```
(describe (add forall ($a $b $c $d $act)
            &ant (     (build synonym *b synonym *c)
                       (build agent *a
                              act (build action *act
                              object *b
                              indobj *d)))
            cq (       build agent *a
                              act (build action *act
                              object *c
                              indobj *d))))
```



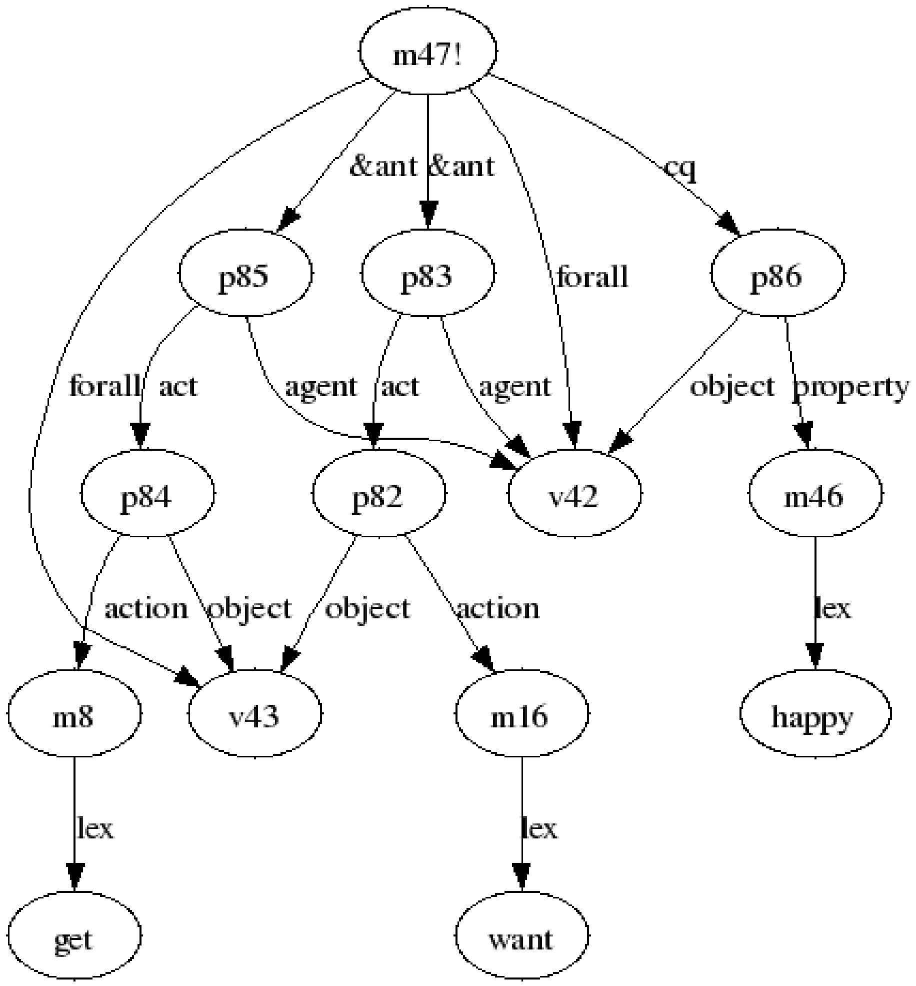
If A takes action ACT directly on C, and B is a member of class C, then A takes action ACT directly on B.

```
(describe (add forall ($a $b $c $act)
          &ant (    (build member *b class *c)
                    (build agent *a
                         act (build action *act
                         object *c)))
          cq (      build agent *a
                         act (build action *act
                         object *b))))
```



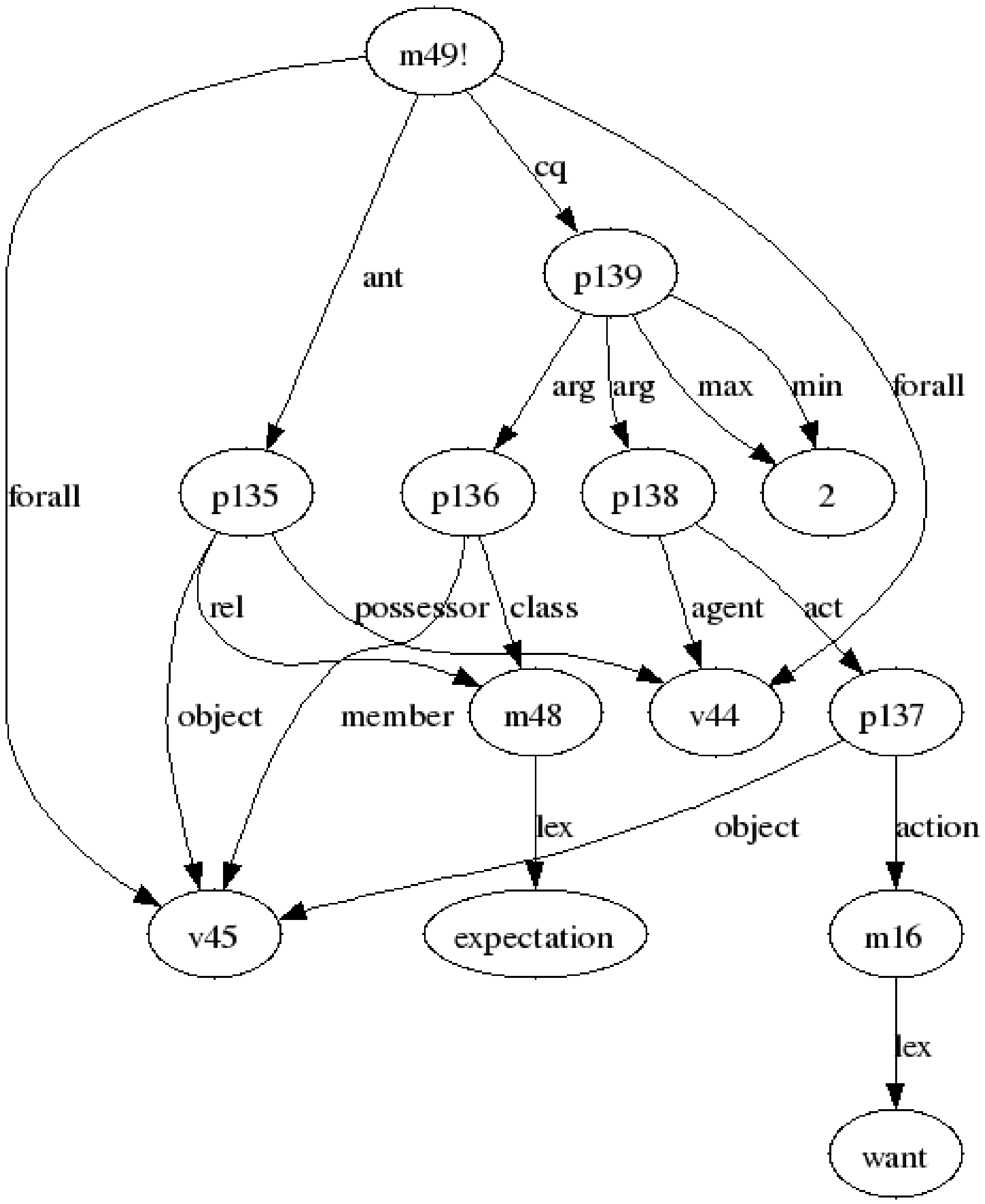
If A wants B, and A gets B, then A is happy.

```
(describe (add forall ($a $b)
        &ant (    (build agent *a
                        act (build action (build lex "want")
                        object *b))
                  (build agent *a
                        act (build action (build lex "get")
                        object *b)))
        cq (      build object *a
                        property (build lex "happy"))))
```



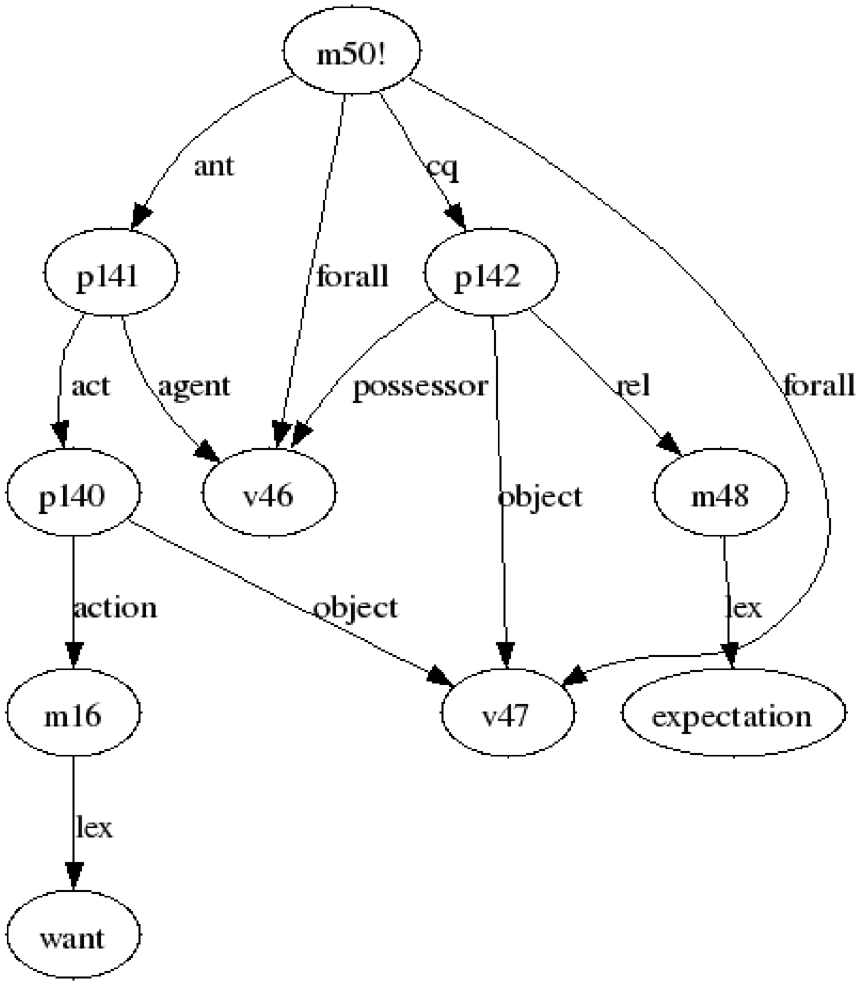
If something E is A's expectation, then E is an expectation, and A wants E.

```
(describe (add forall ($a $e)
          ant (     build object *e
                    rel (build lex "expectation")
                    possessor *a)
          cq (      build min 2 max 2
                    arg ((build member *e
                              class (build lex "expectation"))
                         (build agent *a
                              act (build action (build lex "want")
                              object *e))))))
```



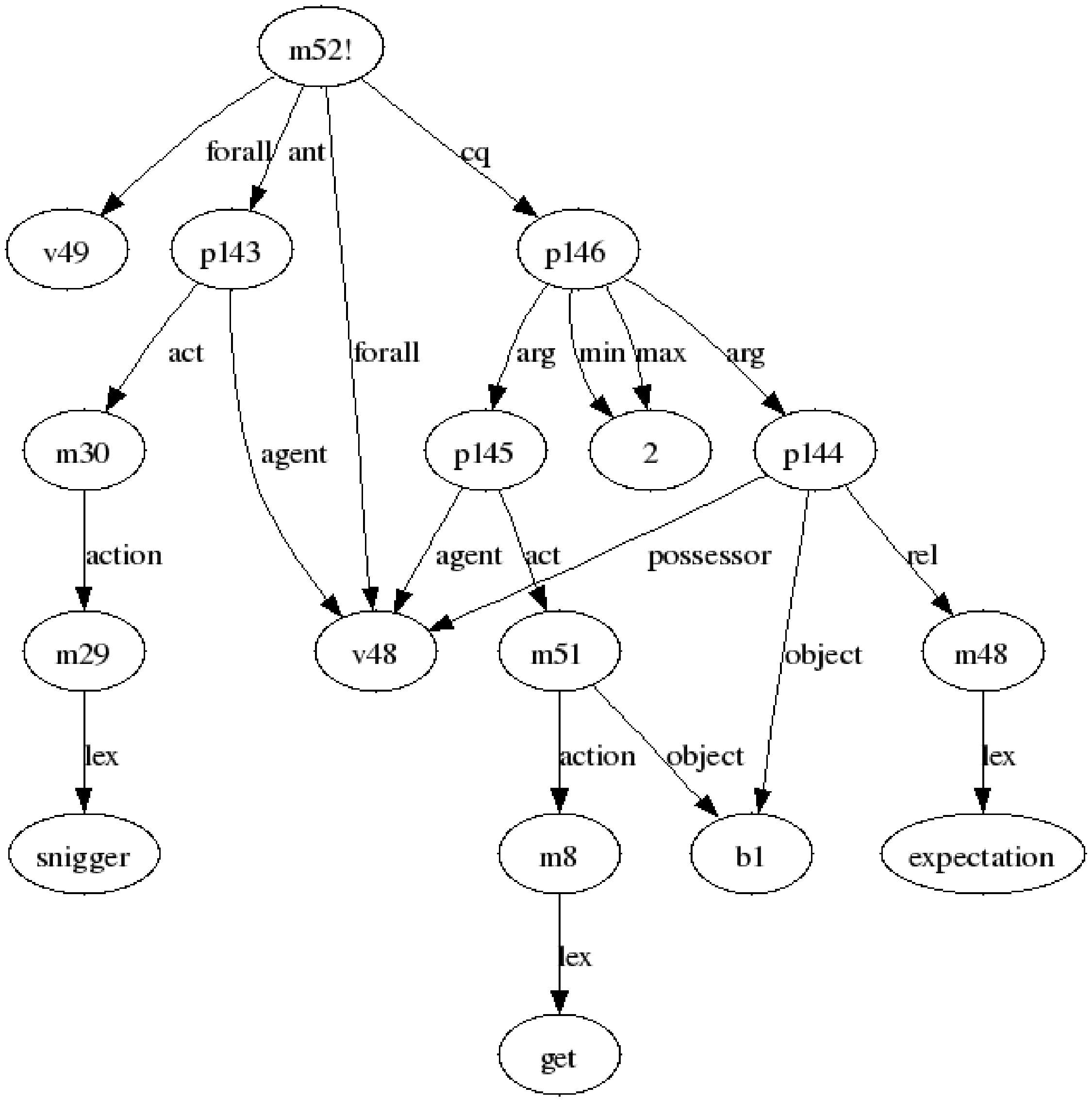
If A wants E, then E is A's expectation.

```
(describe (add forall ($a $e)
          ant (     build agent *a
                         act (build action (build lex "want")
                         object *e))
          cq (      build object *e
                         rel (build lex "expectation")
                         possessor *a)))
```



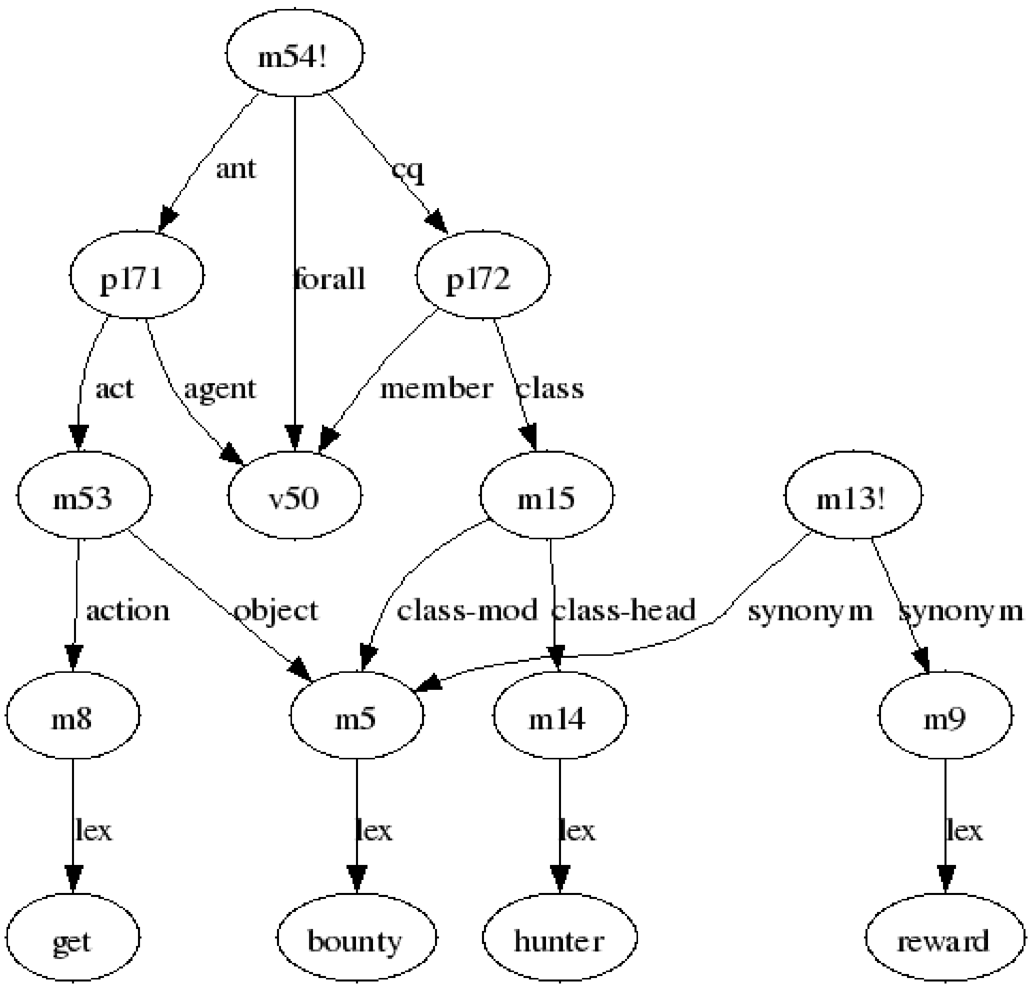
If A sniggers, and something E is A's expectation, then A gets E.

```
(describe (add forall ($a $e)
        ant (     build agent *a
                       act (build action (build lex "snigger")))
        cq (      build min 2 max 2
                  arg (     (build object #expectation
                                 rel (build lex "expectation")
                                 possessor *a)
                            (build agent *a
                                 act (build action (build lex "get")
                                 object *expectation))))))
```



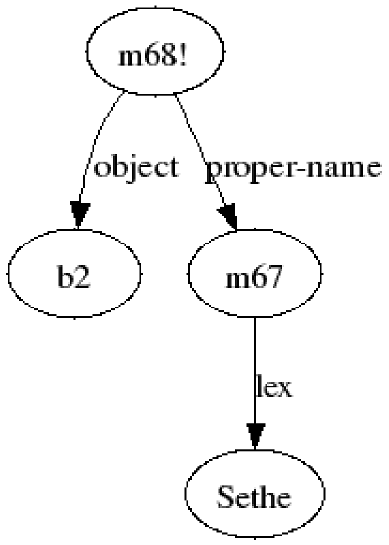
If A gets bounty, then A is a bounty hunter.

```
(describe (add forall $a
          ant (     (build agent *a
                         act (build action (build lex "get")
                         object (build lex "bounty"))))
          cq (      build member *a
                         class (build class-mod (build lex "bounty")
                                   class-head (build lex "hunter")))))
```

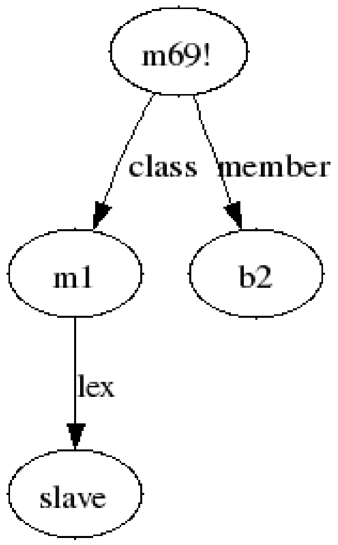


Appendix B: Semantic network diagrams of the passage

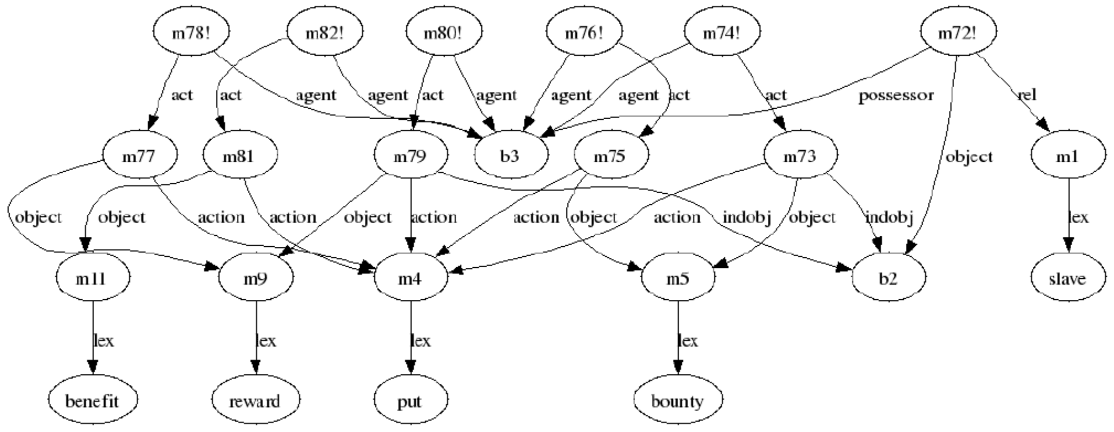
There is someone named Sethe.



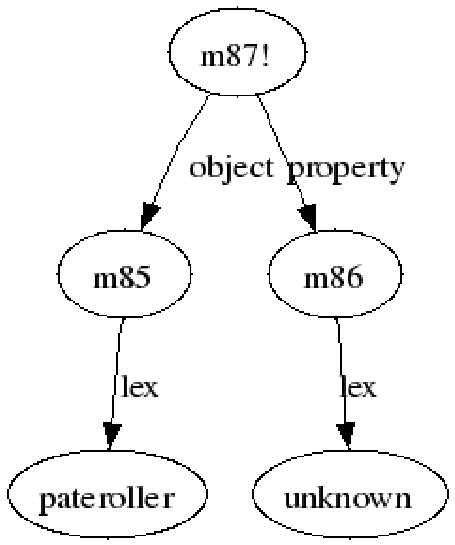
Sethe is a slave.



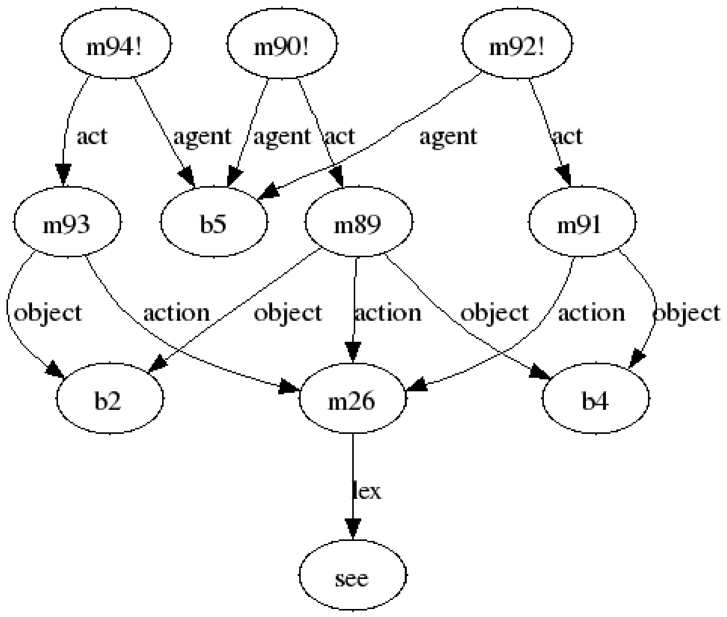
Sethe is the master's slave.



‘Pateroller’ is an unknown word.



Pateroller sees Sethe and Amy.



Appendix C: Script of Running Demo

```
==============================================================
Starting image `/util/acl/composer'
  with no arguments
  in directory `/home/csgrad/nanmeng/'
  on machine `localhost'.

;;; Installing locale patch, version 1.
International Allegro CL Enterprise Edition
8.1 [Linux (x86)] (Oct 27, 2008 11:52)
Copyright (C) 1985-2007, Franz Inc., Oakland, CA, USA.  All Rights Reserved.

This development copy of Allegro CL is licensed to:
   [4549] University at Buffalo

;; Optimization settings: safety 1, space 1, speed 1, debug 2.
;; For a complete description of all compiler switches given the current
;; optimization settings evaluate (explain-compiler-settings).
;;---
;; Current reader case mode: :case-sensitive-lower
cl-user(1): ;; Setting (stream-external-format *terminal-io*) to :utf-8.
cl-user(2): :ld /projects/snwiz/bin/sneps
; Loading /projects/snwiz/bin/sneps.lisp
;;; Installing jlinker patch, version 1.
;;; Installing regexp2-s patch, version 1.
Loading system SNePS...10% 20% 30% 40% 50% 60% 70% 80% 90% 100%
SNePS-2.7 [PL:1 2008/02/12 17:19:45] loaded.
Type `(sneps)' or `(snepslog)' to get started.
cl-user(3): (setf cl-user::*use-gui-show* nil)
nil
cl-user(4): (sneps)


   Welcome to SNePS-2.7 [PL:1 2008/02/12 17:19:45]

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.

   12/6/2008 16:28:51

* (demo "c2")

File /home/csgrad/nanmeng/c2 is now the source of input.

 CPU time : 0.00

* ; =======================================================================
; FILENAME:   c2
; DATE:       from Sep 30 2008 to
; PROGRAMMER: Nan Meng

;; this template version:   snepsul-template.demo-20061005.txt

; 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 "c.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:
;; UNCOMMENT THE ONE YOU *DO* WANT
;; AND DELETE THE OTHER!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
^(
--> load "/projects/rapaport/CVA/STN2/defun_noun.cl")
; Loading /projects/rapaport/CVA/STN2/defun_noun.cl
t

 CPU time : 0.03

* ;; ^(load "~/defun_noun.cl")

; Clear the SNePS network:
(resetnet)

Net reset - Relations and paths are still defined

 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

* ;
; ;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")
^(
--> load "~/rels")
; Loading /home/csgrad/nanmeng/rels
```

```
t

 CPU time : 0.00

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

*
;; Slave masters put bounties on their slaves.
(describe (add forall ($sm $sl)
       &ant ( (build member *sm class (build class-mod (build lex "slave")
                        class-head (build lex "master")))
           (build member *sl class (build lex "slave"))
           (build object *sl rel (build lex "slave") possessor *sm))
       cq (   (build agent *sm
              act (build action (build lex "put")
                  object (build lex "bounty")
                  indobj *sl)))) = master-put-bounty-on-slave)

(m6! (forall v2 v1)
 (&ant (p3 (object v2) (possessor v1) (rel (m1 (lex slave))))
  (p2 (class (m1)) (member v2))
  (p1 (class (m3 (class-head (m2 (lex master))) (class-mod (m1))))
   (member v1)))
 (cq
  (p5
   (act (p4 (action (m4 (lex put))) (indobj v2) (object (m5 (lex bounty)))))
   (agent v1))))

(m6!)

 CPU time : 0.00

*
;; If A puts bounty on something STH and B returns that thing to A, then B gets
rewards (from A).
(describe (add forall ($a $b $sth)
       &ant ( (build agent *a
              act (build action (build lex "put")
                  object (build lex "bounty")
                  indobj *sth))
           (build agent *b
              act (build action (build lex "return")
                  object *sth
                  indobj *a)))
       cq (   (build agent *b
              act (build action (build lex "get")
                  object (build lex "reward")
                  indobj *a)))) = bounty-return-reward)

(m10! (forall v5 v4 v3)
 (&ant
  (p9 (act (p8 (action (m7 (lex return))) (indobj v3) (object v5)))
   (agent v4))
  (p7
   (act (p6 (action (m4 (lex put))) (indobj v5) (object (m5 (lex bounty)))))
   (agent v3)))
 (cq
  (p11
   (act (p10 (action (m8 (lex get))) (indobj v3) (object (m9 (lex reward)))))
   (agent v4))))

(m10!)

 CPU time : 0.00

*
;; Reward and benefit are synonyms.
(describe (add synonym (build lex "reward") synonym (build lex "benefit")))
```

```
(m12! (synonym (m11 (lex benefit)) (m9 (lex reward))))

(m12!)

 CPU time : 0.00

*
;; Bounty and reward are synonyms.
(describe (add synonym (build lex "bounty") synonym (build lex "reward")))

(m13! (synonym (m9 (lex reward)) (m5 (lex bounty))))

(m13!)

 CPU time : 0.00

*
;; Bounty hunters want bounty.
(describe (add forall $b
       ant (  (build member *b class (build class-mod (build lex "bounty")
                          class-head (build lex "hunter"))))
       cq (   (build agent *b
               act (build action (build lex "want")
                   object (build lex "bounty"))))) = bounty-hunter-want-bounty)

(m18! (forall v6)
 (ant
  (p12
   (class (m15 (class-head (m14 (lex hunter))) (class-mod (m5 (lex bounty)))))
   (member v6)))
 (cq (p13 (act (m17 (action (m16 (lex want))) (object (m5)))) (agent v6))))

(m18!)

 CPU time : 0.00

*
;; The concept of bounty hunter, has proper-name "bounty hunter"
(describe (add object (build class-mod (build lex "bounty")
               class-head (build lex "hunter"))
       proper-name (build lex "bounty\ hunter")) = bounty-hunter-proper-name)

(m20!
 (object (m15 (class-head (m14 (lex hunter))) (class-mod (m5 (lex bounty)))))
 (proper-name (m19 (lex bounty hunter))))

(m20!)

 CPU time : 0.01

*
(describe (add object (build class-mod (build lex "slave")
               class-head (build lex "master"))
       proper-name (build lex "slave\ master")) = slave-master-proper-name)

(m22!
 (object (m3 (class-head (m2 (lex master))) (class-mod (m1 (lex slave)))))
 (proper-name (m21 (lex slave master))))

(m22!)

 CPU time : 0.00

*
(describe (add forall ($a $c $n)
       &ant ( (build member *a class *c)
           (build object *c proper-name *n))
       cq (   (build member *a class *n))) = member-class-proper-name)
```

```
(m23! (forall v9 v8 v7)
 (&ant (p15 (object v8) (proper-name v9)) (p14 (class v8) (member v7)))
 (cq (p16 (class v9) (member v7))))
(m22!
 (object (m3 (class-head (m2 (lex master))) (class-mod (m1 (lex slave)))))
 (proper-name (m21 (lex slave master))))
(m20!
 (object (m15 (class-head (m14 (lex hunter))) (class-mod (m5 (lex bounty)))))
 (proper-name (m19 (lex bounty hunter))))

(m23! m22! m20!)

 CPU time : 0.00

*
;; If A is a bounty hunter, and B puts bounty on C, and A catches C, then A returns
C to B (and gets bounty from B).
(describe (add forall ($a $b $c)
       &ant ( (build member *a class (build class-mod (build lex "bounty")
                           class-head (build lex "hunter")))
           (build agent *b
              act (build action (build lex "put")
                  object (build lex "bounty")
                  indobj *c))
           (build agent *a
              act (build action (build lex "catch")
                  object *c)))
       cq (   (build agent *a
              act (build action (build lex "return")
                  object *c
                  indobj *b)))) = bounty-hunter-catch-return)

(m25! (forall v12 v11 v10)
 (&ant (p21 (act (p20 (action (m24 (lex catch))) (object v12))) (agent v10))
  (p19
   (act (p18 (action (m4 (lex put))) (indobj v12) (object (m5 (lex bounty)))))
   (agent v11))
  (p17 (class (m15 (class-head (m14 (lex hunter))) (class-mod (m5))))
   (member v10)))
 (cq
  (p23 (act (p22 (action (m7 (lex return))) (indobj v11) (object v12)))
   (agent v10))))

(m25!)

 CPU time : 0.00

*
;; If A is a bounty hunter, and B puts bounty on C, and A sees C, then A catches
C.
(describe (add forall ($a $b $c)
       &ant ( (build member *a class (build class-mod (build lex "bounty")
                           class-head (build lex "hunter")))
           (build agent *b
              act (build action (build lex "put")
                  object (build lex "bounty")
                  indobj *c))
           (build agent *a
              act (build action (build lex "see")
                  object *c)))
       cq (   (build agent *a
              act (build action (build lex "catch")
              object *c)))) = bounty-hunter-see-catch)

(m27! (forall v15 v14 v13)
 (&ant (p30 (act (p29 (action (m26 (lex see))) (object v15))) (agent v13))
  (p28
   (act (p27 (action (m4 (lex put))) (indobj v15) (object (m5 (lex bounty)))))
   (agent v14))
  (p26 (class (m15 (class-head (m14 (lex hunter))) (class-mod (m5))))
   (member v13)))
 (cq (p32 (act (p31 (action (m24 (lex catch))) (object v15))) (agent v13))))

(m27!)

 CPU time : 0.00

*
;; If A returns C to B, then B gets C, and B gets C from A.
(describe (add forall ($a $b $c)
        ant (  (build agent *a
                act (build action (build lex "return")
                    object *c
                    indobj *b)))
        cq (   build min 2 max 2
            arg (  (build agent *b
                    act (build action (build lex "get")
                        object *c))
                (build agent *b
                    act (build action (build lex "get")
                        object *c
                        indobj *a))))) = return-get-from)

(m28! (forall v18 v17 v16)
 (ant
  (p34 (act (p33 (action (m7 (lex return))) (indobj v17) (object v18)))
   (agent v16)))
 (cq
  (p39 (min 2) (max 2)
   (arg
    (p38 (act (p37 (action (m8 (lex get))) (indobj v16) (object v18)))
     (agent v17))
    (p36 (act (p35 (action (m8)) (object v18))) (agent v17))))))

(m28!)

 CPU time : 0.00

*
;; If A sniggers, then A is happy (smug).
(describe (add forall $a
        ant (  build agent *a
                act (build action (build lex "snigger")))
        cq (   (build object *a property (
                            (build lex "smug")
                            (build lex "mean"))))))

(m33! (forall v19)
 (ant (p40 (act (m30 (action (m29 (lex snigger))))) (agent v19)))
 (cq (p41 (object v19) (property (m32 (lex mean)) (m31 (lex smug))))))

(m33!)

 CPU time : 0.01

*
(describe (add forall ($a $b)
        &ant ( (build agent *a
                act (build action (build lex "return")
                        object *b))
            (build member *b class (build lex "slave")))
        cq (   (build object *a property (build lex "evil")))))

(m35! (forall v21 v20)
 (&ant (p44 (class (m1 (lex slave))) (member v21))
  (p43 (act (p42 (action (m7 (lex return))) (object v21))) (agent v20)))
 (cq (p45 (object v20) (property (m34 (lex evil))))))
```

```
(m35!)

 CPU time : 0.00

*
;; If A sniggers, then A laughs for him/herself.
(describe (add forall $a
       ant (  (build agent *a
              act (build action (build lex "snigger"))))
       cq (   (build agent *a
              act (build action (build lex "laugh")
                  indobj *a)))) = snigger-laugh-for)

(m37! (forall v22)
 (ant (p48 (act (m30 (action (m29 (lex snigger))))) (agent v22)))
 (cq (p50 (act (p49 (action (m36 (lex laugh))) (indobj v22))) (agent v22))))

(m37!)

 CPU time : 0.01

*
;; If A laughs for him/herself, then A gets some benefit.
(describe (add forall $a
       ant (  (build agent *a
              act (build action (build lex "laugh")
                  indobj *a)))
       cq (   build agent *a
              act (build action (build lex "get")
                  object (build lex "benefit")))) = laugh-for-get-benefit)

(m39! (forall v23)
 (ant (p52 (act (p51 (action (m36 (lex laugh))) (indobj v23))) (agent v23)))
 (cq
  (p53 (act (m38 (action (m8 (lex get))) (object (m11 (lex benefit)))))
   (agent v23))))

(m39!)

 CPU time : 0.01

*
;; If A takes action ACT directly on B, B and C are synonyms, then A takes action
ACT directly on C.
(describe (add forall ($a $b $c $act)
       &ant ( (build synonym *b synonym *c)
           (build agent *a
              act (build action *act
                  object *b)))
       cq (   build agent *a
              act (build action *act
                  object *c))) = action-synonym-direct)

(m40! (forall v27 v26 v25 v24)
 (&ant (p56 (act (p55 (action v27) (object v25))) (agent v24))
  (p54 (synonym v26 v25)))
 (cq (p58 (act (p57 (action v27) (object v26))) (agent v24))))
(m13! (synonym (m9 (lex reward)) (m5 (lex bounty))))
(m12! (synonym (m11 (lex benefit)) (m9)))

(m40! m13! m12!)

 CPU time : 0.00

*
;; If A takes action ACT directly on B indirectly on D, B and C are synonyms,
then A takes action ACT directly on C indirectly on D.
(describe (add forall ($a $b $c $d $act)
        &ant ( (build synonym *b synonym *c)
            (build agent *a
                act (build action *act
                    object *b
                    indobj *d)))
        cq (    build agent *a
                act (build action *act
                    object *c
                    indobj *d))) = action-synonym-direct-indirect)

(m42! (forall v32 v30 v29 v28)
 (&ant (p65 (act (p64 (action v32) (object v29))) (agent v28))
  (p59 (synonym v30 v29)))
 (cq (p67 (act (p66 (action v32) (object v30))) (agent v28))))
(m41! (forall v32 v31 v30 v29 v28)
 (&ant (p61 (act (p60 (action v32) (indobj v31) (object v29))) (agent v28))
  (p59))
 (cq (p63 (act (p62 (action v32) (indobj v31) (object v30))) (agent v28))))
(m40! (forall v27 v26 v25 v24)
 (&ant (p56 (act (p55 (action v27) (object v25))) (agent v24))
  (p54 (synonym v26 v25)))
 (cq (p58 (act (p57 (action v27) (object v26))) (agent v24))))
(m13! (synonym (m9 (lex reward)) (m5 (lex bounty))))
(m12! (synonym (m11 (lex benefit)) (m9)))

(m42! m41! m40! m13! m12!)

 CPU time : 0.01

*
;; If A takes action ACT directly on C, and B is a member of class C, then A
takes action ACT directly on B.
(describe (add forall ($a $b $c $act)
        &ant ( (build member *b class *c)
            (build agent *a
                act (build action *act
                    object *c)))
        cq (    build agent *a
                act (build action *act
                    object *b))) = action-member-direct)

(m43! (forall v36 v35 v34 v33)
 (&ant (p70 (act (p69 (action v36) (object v35))) (agent v33))
  (p68 (class v35) (member v34)))
 (cq (p72 (act (p71 (action v36) (object v34))) (agent v33))))

(m43!)

CPU time : 0.00

*
;; If A takes action ACT directly on C indirectly on D, and B is a member of
class C, then A takes action ACT directly on B indirectly on D.
(describe (add forall ($a $b $c $d $act)
        &ant ( (build member *b class *c)
            (build agent *a
                act (build action *act
                    object *c
                    indobj *d)))
        cq (    build agent *a
                act (build action *act
                    object *b
                    indobj *d))) = action-member-direct-indirect)

(m45! (forall v41 v39 v38 v37)
 (&ant (p79 (act (p78 (action v41) (object v39))) (agent v37))
  (p73 (class v39) (member v38)))
 (cq (p81 (act (p80 (action v41) (object v38))) (agent v37))))
(m44! (forall v41 v40 v39 v38 v37)
 (&ant (p75 (act (p74 (action v41) (indobj v40) (object v39))) (agent v37))
  (p73))
 (cq (p77 (act (p76 (action v41) (indobj v40) (object v38))) (agent v37))))
(m43! (forall v36 v35 v34 v33)
 (&ant (p70 (act (p69 (action v36) (object v35))) (agent v33))
  (p68 (class v35) (member v34)))
 (cq (p72 (act (p71 (action v36) (object v34))) (agent v33))))

(m45! m44! m43!)

 CPU time : 0.01

*
;; If A wants B, and A gets B, then A is happy.
(describe (add forall ($a $b)
       &ant ( (build agent *a
              act (build action (build lex "want")
                  object *b))
           (build agent *a
              act (build action (build lex "get")
                  object *b)))
       cq (   build object *a property (build lex "happy"))) = want-get-happy)

(m47! (forall v43 v42)
 (&ant (p85 (act (p84 (action (m8 (lex get))) (object v43))) (agent v42))
  (p83 (act (p82 (action (m16 (lex want))) (object v43))) (agent v42)))
 (cq (p86 (object v42) (property (m46 (lex happy))))))

(m47!)

 CPU time : 0.05

*
;; If something E is A's expectation, then E is an expectation, and A wants E.
(describe (add forall ($a $e)
       ant (  build object *e rel (build lex "expectation") possessor *a)
       cq (   build min 2 max 2
           arg (  (build member *e class (build lex "expectation"))
              (build agent *a
                  act (build action (build lex "want")
                      object *e))))) = expect-want)

(m49! (forall v45 v44)
 (ant (p135 (object v45) (possessor v44) (rel (m48 (lex expectation)))))
 (cq
  (p139 (min 2) (max 2)
   (arg (p138 (act (p137 (action (m16 (lex want))) (object v45))) (agent v44))
    (p136 (class (m48)) (member v45))))))

(m49!)

 CPU time : 0.00

*
;; If A wants E, then E is A's expectation.
(describe (add forall ($a $e)
       ant (  build agent *a
              act (build action (build lex "want")
                  object *e))
       cq (   build object *e rel (build lex "expectation") possessor *a)) =
want-expect)

(m50! (forall v47 v46)
 (ant (p141 (act (p140 (action (m16 (lex want))) (object v47))) (agent v46)))
 (cq (p142 (object v47) (possessor v46) (rel (m48 (lex expectation))))))

(m50!)

 CPU time : 0.01
```

```
*
;; If A sniggers, and something E is A's expectation, then A gets E.
(describe (add forall ($a $e)
       ant (  build agent *a
              act (build action (build lex "snigger")))
       cq (   build min 2 max 2
           arg (  (build object #expectation rel (build lex "expectation")
possessor *a)
              (build agent *a
                 act (build action (build lex "get")
                    object *expectation))))) = snigger-get-expect)

(m52! (forall v49 v48)
 (ant (p143 (act (m30 (action (m29 (lex snigger))))) (agent v48)))
 (cq
  (p146 (min 2) (max 2)
   (arg (p145 (act (m51 (action (m8 (lex get))) (object b1))) (agent v48))
    (p144 (object b1) (possessor v48) (rel (m48 (lex expectation))))))))

(m52!)

 CPU time : 0.03

*
;; If A gets bounty, then A is a bounty hunter.
(describe (add forall $a
       ant (  (build agent *a
              act (build action (build lex "get")
                 object (build lex "bounty"))))
       cq (   build member *a class (build class-mod (build lex "bounty")
                    class-head (build lex "hunter")))) =
want-bounty-bounty-hunter)

(m54! (forall v50)
 (ant
  (p171 (act (m53 (action (m8 (lex get))) (object (m5 (lex bounty)))))
   (agent v50)))
 (cq
  (p172 (class (m15 (class-head (m14 (lex hunter))) (class-mod (m5))))
   (member v50))))
(m13! (synonym (m9 (lex reward)) (m5)))

(m54! m13!)

 CPU time : 0.10

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

;; Sethe is a runaway female slave.

; There is someone named Sethe.
(describe (add object #sethe proper-name (build lex "Sethe")) =
someone-is-named-sethe)

(m68! (object b2) (proper-name (m67 (lex Sethe))))

(m68!)

 CPU time : 0.02

*
; Someone is named John.
(describe (add object #john proper-name (build lex "John")))

(m70! (object b3) (proper-name (m69 (lex John))))
```

```
(m70!)

 CPU time : 0.00

*
; Sethe is a slave.
(describe (add member *sethe class (build lex "slave")) = sethe-is-a-slave)

(m71! (class (m1 (lex slave))) (member b2))

(m71!)

 CPU time : 0.01

*
; Someone is a slave master.
(describe (add member #master
       class (build class-mod (build lex "slave")
              class-head (build lex "master"))) = someone-is-a-slavemaster)

(m73! (class (m21 (lex slave master))) (member b4))
(m72! (class (m3 (class-head (m2 (lex master))) (class-mod (m1 (lex slave)))))
 (member b4))

(m73! m72!)

 CPU time : 0.00

*
; Sethe is the master's slave.
(describe (add object *sethe rel (build lex "slave") possessor *master) =
sethe-is-masters-slave)

(m84! (act (m83 (action (m4 (lex put))) (object (m11 (lex benefit)))))
 (agent b4))
(m82! (act (m81 (action (m4)) (indobj b2) (object (m9 (lex reward)))))
 (agent b4))
(m80! (act (m79 (action (m4)) (object (m9)))) (agent b4))
(m78! (act (m77 (action (m4)) (object (m5 (lex bounty))))) (agent b4))
(m76! (act (m75 (action (m4)) (indobj b2) (object (m5)))) (agent b4))
(m74! (object b2) (possessor b4) (rel (m1 (lex slave))))

(m84! m82! m80! m78! m76! m74!)

 CPU time : 0.01

*
;; A pateroller passing would have sniggered to see two throwaway people, two
lawless outlaws --- a slave and a barefoot white woman with unpinned hair ---
wrapping a ten-minute-old baby in the rags they wore.

; There is someone named Amy
(describe (add object #amy proper-name (build lex "Amy")))

(m86! (object b5) (proper-name (m85 (lex Amy))))

(m86!)

 CPU time : 0.01

*
; "pateroller" is an unknown word.
(describe (add object (build lex "pateroller") property (build lex "unknown")))

(m89! (object (m87 (lex pateroller))) (property (m88 (lex unknown))))

(m89!)

 CPU time : 0.00
```

```
*
; Someone is a pateroller.
(describe (add member #pt class (build lex "pateroller")))

(m90! (class (m87 (lex pateroller))) (member b6))

(m90!)

 CPU time : 0.00

*
(describe (add agent *pt
       act (build action (build lex "see")
           object (*sethe *amy))))

(m96! (act (m95 (action (m26 (lex see))) (object b2))) (agent b6))
(m94! (act (m93 (action (m26)) (object b5))) (agent b6))
(m92! (act (m91 (action (m26)) (object b5 b2))) (agent b6))

(m96! m94! m92!)

 CPU time : 0.01

*
; If a pateroller A sees Sethe and Amy, then A will snigger.
(describe (add forall $p
       &ant ( (build member *p class (build lex "pateroller"))
           (build agent *p
           act (build action (build lex "see")
               object (*sethe *amy))))
       cq (   (build agent *p
               act (build action (build lex "snigger"))))))

(m126! (min 2) (max 2)
 (arg (m125 (class (m48 (lex expectation))) (member (m11 (lex benefit))))
  (m121! (act (m120 (action (m16 (lex want))) (object (m11)))) (agent b6))))
(m124! (min 2) (max 2)
 (arg (m123 (class (m48)) (member (m9 (lex reward))))
  (m117! (act (m116 (action (m16)) (object (m9)))) (agent b6))))
(m122! (object (m11)) (possessor b6) (rel (m48)))
(m119! (object (m9)) (possessor b6) (rel (m48)))
(m118! (min 2) (max 2)
 (arg (m110! (act (m17 (action (m16)) (object (m5 (lex bounty))))) (agent b6))
  (m61! (class (m48)) (member (m5)))))
(m115! (object b6) (property (m46 (lex happy))))
(m114! (object (m5)) (possessor b6) (rel (m48)))
(m113! (act (m112 (action (m24 (lex catch))) (object b2))) (agent b6))
(m111! (class (m19 (lex bounty hunter))) (member b6))
(m109! (class (m15 (class-head (m14 (lex hunter))) (class-mod (m5))))
 (member b6))
(m108! (act (m53 (action (m8 (lex get))) (object (m5)))) (agent b6))
(m107! (act (m106 (action (m8)) (object (m9)))) (agent b6))
(m105! (act (m38 (action (m8)) (object (m11)))) (agent b6))
(m104! (min 2) (max 2)
 (arg (m103 (object b1) (possessor b6) (rel (m48)))
  (m102 (act (m51 (action (m8)) (object b1))) (agent b6))))
(m101! (act (m100 (action (m36 (lex laugh))) (indobj b6))) (agent b6))
(m99! (object b6) (property (m32 (lex mean)) (m31 (lex smug))))
(m98! (act (m30 (action (m29 (lex snigger))))) (agent b6))
(m97! (forall v51)
 (&ant (p294 (act (m91 (action (m26 (lex see))) (object b5 b2))) (agent v51))
  (p293 (class (m87 (lex pateroller))) (member v51)))
 (cq (p295 (act (m30)) (agent v51))))
(m92! (act (m91)) (agent b6))
(m90! (class (m87)) (member b6))

(m126! m124! m122! m121! m119! m118! m117! m115! m114! m113! m111! m110! m109!
 m108! m107! m105! m104! m101! m99! m98! m97! m92! m90! m61!)
 CPU time : 0.19

*
; Sethe and Amy are outlaws.
(describe (add member (*sethe *amy) class (build lex "outlaw")))

(m130! (class (m127 (lex outlaw))) (member b5))
(m129! (class (m127)) (member b2))
(m128! (class (m127)) (member b5 b2))

(m130! m129! m128!)

 CPU time : 0.00

*
^(
--> defineNoun "pateroller")
8628736 bytes have been tenured, next gc will be global.
See the documentation for variable excl:*global-gc-behavior* for more
information.
 Definition of pateroller:
 Possible Class Inclusions: m15, bounty hunter,
 Possible Actions: see bounty, see outlaw slave, see outlaw, snigger, laugh,
catch bounty, catch outlaw slave, return bounty, return outlaw slave, get b1,
get bounty, get benefit, get reward, get outlaw slave, want b1, want bounty,
want benefit, want reward,
 Possible Properties: happy, smug, mean,
nil

 CPU time : 2.04

*

End of /home/csgrad/nanmeng/c2 demonstration.

 CPU time : 2.57

*
```

References

Rapaport, William J. (2005), "In Defense of Contextual Vocabulary Acquisition: How to Do Things with Words in Context", in A. Dey et al. (eds.), *Proceedings of the 5th International and Interdisciplinary Conference on Modeling and Using Context (Context-05)* (Berlin: Springer-Verlag Lecture Notes in Artificial Intelligence 3554): 396-409.

Shapiro, Stuart C. and Rapaport, William J. (1995), "An Introduction to a Computational Reader of Narrative", in Judith Felson Duchan, Gail A. Bruder, & Lynne E. Hewitt (eds.), *Deixis in Narrative: A Cognitive Science Perspective* (Hillsdale, NJ: Lawrence Erlbaum Associates): 79-105.

Goldfain, Albert (2003), "Computationally Defining "Harbinger" via Contextual Vocabulary Acquisition", [http://www.cse.buffalo.edu/~rapaport/CVA/Harbinger/harbinger.html]

Xu, Jun (2004), "Computationally Defining 'Ceilidh' from Contextual Cues", [http://www.cse.buffalo.edu/~rapaport/CVA/ceilidh.html]