The Verb Algorithm in the SNePS Rational Engine

Joe Salazar
CSE 727: Seminar on Contextual Vocabulary Acquisition
May 15, 2007

Abstract

Contextual Vocabulary Acquisition (CVA) is the act of deriving the meaning of a word from the context in which it appears. The goal of the CVA project is to develop a computational model of this task. To accomplish this, the SNePS knowledge representation and reasoning system is used to model both the mind of a reader and the process of learning a new word. A cognitive agent known as Cassie is given a representation of a passage and an appropriate set of background knowledge. She then comes up with an approximate definition using reasoning and a CVA algorithm designed for a particular part of speech. In this paper, a revision to the verb algorithm using a package known as the SNePS Rational Engine is introduced. The benefits of this approach are discussed and the new algorithm is tested by comparing its performance against the results of a previous version of the verb algorithm.

Introduction to Contextual Vocabulary Acquisition

Suppose a person who is reading a book comes across a word that they are unfamiliar with. If they decide to try to figure out what its meaning is from the context in which it appears, that reader would be engaging in what is known as contextual vocabulary acquisition (CVA). More formally, CVA is defined as "the active, deliberate acquisition of word meanings from text by reasoning from contextual clues, prior knowledge, language knowledge and hypotheses from prior encounters with the word, but without external sources of help such as dictionaries or people." (Rapaport and Kibby 2002: 3) With repeated exposure to the word in varying contexts, the meaning which the reader has derived will eventually "converge" to a dictionary-like definition. (Rapaport and Ehrlich 2000: 5)

The goal of the CVA project is to develop a computational theory of vocabulary acquisition. By designing algorithms to perform this task, a better understanding of the role of context can be gained and can serve as the foundation for a curriculum in CVA. (Rapaport and Kibby 2002: 3) As things currently stand, algorithms to define nouns and verbs have been produced and are now in the process of being tested. These algorithms are implemented in Lisp and are designed to work with a semantic network in order to properly model the process of learning a new word. The name of the particular semantic network in use is SNePS.

SNePS is an "intensional, propositional, semantic-network knowledge-representation and reasoning system used for research in artificial intelligence and in cognitive science." (Shapiro and Rapaport 1995: 79) The structure and features of SNePS lend themselves well to the task of modeling cognitive processes. For instance, components such as the SNePS Inference Package (SNIP) and the SNePS Belief Revision package (SNeBR) allow a SNePS cognitive agent to

reason with the information present in the network. However, the bulk of the work in this project is done using the SNePS Rational Engine, SNeRE.

SNeRE is a package in SNePS which allows cognitive agents to engage in acting. At the primitive level, acts are written in Lisp code whereas more complex acts are written in terms of other acts. There is also a set of predefined actions which are fairly powerful in a computational sense. For example, the “withall” command takes a set of variables, a proposition which contains those free variables and a call to an action which (presumably) uses those variables. This command will then perform that action for every configuration of variables which satisfies the proposition. Another example of SNeRE’s power is the fact that sequence, selection and iteration all have acting analogues in SNeRE.

The current implementations of the noun and verb algorithms model a process known as “implicit” CVA. By calling a Lisp function after the network is formed, little to no reasoning using the information in the network occurs. Instead, the network is simply “raided” for information which is then processed and delivered to the user. In contrast, an implementation of the verb algorithm in SNeRE would allow much of the information gathering process to be constructed as acts which are carried out by a cognitive agent. An algorithm written in terms of deliberate acts would be performing “explicit” CVA.

Functionality of the New Algorithm

While this version of the verb algorithm was in development, a decision was made to limit the amount of pure Lisp code present as much as possible. The point of this self-imposed constraint was to force most of the code to be written as SNeRE acts. This has the benefit of

reducing most of the verb algorithm to actions performed by a cognitive agent, which may be more useful to researchers who are trying to determine techniques for CVA.

The complex acts which describe how Cassie is to gather the information are dynamically defined using universally quantified rules. Once a verb is identified as the target verb using the member/class case frame, these rules will trigger and the complex acts will become defined in terms of the verb. Once this is done, the DefineVerb primitive action can be used to print out the "verb frame."

The following sections explain what each heading of the verb frame represents and discuss how Cassie derives the information that each component requires. The actual code for each section can be seen in Appendix B. The semantics for each case frame that is used by the algorithm can be found in Appendix A.

Verb

The verb heading is just the name of the verb which is being defined. This information is initially provided to the algorithm when the user invokes the perform command on DefineVerb to get it started. Instead of searching the network to find the verb, the given value is simply reprinted under the *lex* heading of the verb frame.

Transitivity

The transitivity of a verb is directly related to the number of objects that a given verb interacts with. There are typically three classes of transitivity: ditransitive, transitive and intransitive. Ditransitive verbs deal with two objects, one named the "direct" object and one named the "indirect" object. Transitive verbs do not deal with an indirect object and interact only with a direct object. Intransitive verbs do not deal with any objects at all.

This information is implicitly present in the network by the manner in which the sentence is represented. Different SNePSLOG frames are used depending on the transitivity class to which the verb belongs. Consequently, all that is required to determine the class of the verb is to find out which frame is being used to represent the action.

In order to do this, SNeRE's withall command is used to check if any instances of the ditransitive action frame containing the verb exist in the network. If so, then the verb is classified as ditransitive. If not, then the withall command is used again to check if the transitive action frame is in use. If this is the case then the verb is classified as transitive, otherwise it is classified as intransitive.

This information is recorded directly in the network using a "verb-feature" frame. Doing so allows us to avoid repeatedly performing the three-step check when subsequent parts of the program need to know the verb's transitivity. Once the verb's transitivity has been recorded, it is then printed out in the verb frame.

Properties

In several contexts, it can be determined that a given verb has certain properties even though it is not an "object" in the traditional sense. For instance, the verb is frequently given the property of being unknown as it is the word we are looking to define in the first place.

Determining the properties of the verb is fairly easy. By using the "tell-ask" interface, Cassie can perform deductions while inside a primitive action. This allows her to determine what the properties of the verb are by searching the network for instances of the object-property frame with the verb as the object. She then returns a list of all of the properties which is then printed to the screen.

Cause/Effect

This was an interesting case to handle as the cause and effect relationship is not formally defined in any of the available case frame dictionaries. In Becker's algorithm, two types of causes and effects called "direct" and "action" are checked. The difference between the two is the way in which the cause and effect frame is used to build the relationship. For the action causes and effects, both slots of the cause and effect frame are filled with instances of the agent/act/action case frame. The algorithm checks to see if any agent performing the action corresponding to the verb is the cause or effect of some other agent performing another action. For direct causes and effects, the verb itself directly occupies either the cause or effect slot while the other slot is occupied by the agent/act/action case frame. In this case, the algorithm checks to see if the verb action directly causes or is the effect of an agent performing another action.

To determine if the verb is contained within any cause and effect relationships, the algorithm checks all of the aforementioned configurations of this frame using the withall action. If any matches are found, they are then passed to a primitive action which retrieves the action from each of these instances and prints it out.

Object and Agent

Both the object and agent parts of the frame are relatively simple. In Becker's algorithm, these frame slots contained the most common superclass of both the agent and the object. However, this is difficult to do without resorting to the use of list processing functions in Lisp. Therefore, all of the class memberships of the object and the agent are printed in this particular implementation.

Before the DefineVerb action is performed, special paths are defined for the member/class and subclass/superclass relations. This allows the inference package to find all of the class memberships by following any subclass/superclass chains that exist in the network.

Once this is done, then the class memberships of every agent and every object associated with the verb are printed to the screen.

Similar Actions

There are two different categories which are considered under the heading of similar actions. The first are any terms which are related to the verb through an “equivalency” relation. If the verb appears in the network within the Equivalent, Similar or are-synonymous case frames then the algorithm will display the terms that the verb is connected to. If no such information exists then the algorithm will display a list of actions performed on the same object on which the verb performed.

In Becker’s algorithm, there is another type of similar action – actions performed with the same instrument that was associated with the verb. Only the former is searched for in this implementation since support for instruments is not available at this time.

Testing and Results

The primary test of this algorithm was a comparison of results between Becker’s algorithm and the SNeRE algorithm using Becker’s demo of the verb “perambulate.” This demo is divided into different stages, and at the end of each stage the verb algorithm is executed. This is designed to show how the definition evolves with exposure to different contexts. Since the definitions change with every stage, this is an excellent test for the algorithm.

The context for the first stage was the sentence "In the morning we rose to *perambulate* a city and surveyed the ruins of ancient magnificence..." (Becker 2005: 15) The following is a comparison of the resulting outputs:

Becker's Algorithm	SNeRE Algorithm
lex: perambulate; property: unknown; similar action: (rose survey) transitivity: (transitive) object: (city) agent: (person)	perform DefineVerb(perambulate) Verb: perambulate Class: (verb) Transitivity: (transitive) Properties: (unknown) Similar Actions: nil Agent: (person) Object: (place city)

Table 1: The outputs of both algorithms after the first context.

The similar actions output in the first algorithm are not detected in the SNeRE algorithm since they are not explicitly said to be similar and are not actions performed on the same object that the verb acts on. However, an interesting bug was found at this stage. Each slot in the frame can be printed out independently by executing the perform command on the action (primitive or complex) representing that particular frame slot. This is particularly useful for debugging as the entire DefineVerb command does not need to be executed to test individual parts of the frame. However, performing the DefineSimilarActions command independently results in the verb itself being returned as a similar action. The reason for the difference in functionality has not yet been determined.

The context for the second sentence was the sentence "We are going to attack a city and destroy it soon." (Becker 2005: 16) The following are the outputs:

Becker's Algorithm	SNeRE Algorithm
lex: perambulate; property: unknown; similar action: (unmake activity destroy attack) transitivity: (transitive) object: (city) agent: (person)	Verb: perambulate Class: (verb) Transitivity: (transitive) Properties: (unknown) Similar Actions:(destroy attack) Agent: (person) Object: (place city)

Table 2: The outputs of both algorithms after the second context.

Both algorithms have found some common ground when it comes to the list of similar actions. While the superclasses of “destroy” and “attack” are not included in the SNeRE algorithm at this particular stage, the algorithms seem to agree on all of the other information present in the verb frame.

The context for the third sentence was “Nightly did the hereditary prince of the land perambulate the streets of his capital…” (Becker 2005: 16) The outputs are as follows:

Becker’s Algorithm	SNeRE Algorithm
lex: perambulate; property: unknown; similar action: (unmake activity destroy attack) transitivity: (transitive) object: (city) agent: (person)	perform DefineVerb(perambulate) Verb: perambulate Class: (verb) Transitivity: (transitive) Properties: (unknown) Similar Actions:(attack unmake activity destroy) Agent: (person) Agent: (person) Object: (streets place) Object: (place city)

Table 3: The outputs of both algorithms after the third context.

At this point an issue with the SNeRE implementation presented itself. Since the withall command is being used, the agent and object frames are printing out multiple times for every agent which performs the verb action and every object that the verb acts upon. While this is not a frequent occurrence in the context tests that are being run, a strategy for dealing with this is still necessary as the process of CVA is intended to work over varying contexts. Printing out as many independent copies of the agent and object frames as there are contexts is not a palatable solution.

Another problem which also showed up at this stage was the fact that producing the similar actions list took a non-trivial amount of time. This indicates that the way in which support for similar actions is implemented is probably flawed. There may be the potential for infinite loop situations with logical inferences or very wasteful searches through the network.

There are three more stages to the demo for “perambulate” but the first stages have sufficiently demonstrated the capabilities of the SNeRE algorithm. Many of the verb frame components work perfectly fine but the tests have clearly shown that other parts (such as similar action) need serious revision.

Future Work

One of the major difficulties of using SNePSLOG for this project is that all case frames have to be defined before they can be used. This presented a problem during the conversion process as Becker’s algorithm is constructed almost entirely in terms of arc labels and paths. There was no information available to determine what case frame a particular arc belonged to. In most of these cases, support for that particular part of the algorithm was not implemented. Anyone who continues work on this project should be aware that support needs to be built in for many more case frames.

Furthermore, many of the case frames that newer versions of the verb algorithm deal with do not have their semantics formally defined. The most common frames (such as those shared with the noun algorithm) have had their representational quality discussed at length and have solid linguistic justification for the manner in which they are constructed. However, there are some very important frames (such as those dealing with instruments) which simply haven’t been scrutinized yet. They need to be analyzed further to ensure that they represent information in a sensible manner. Once this is done, they can then be added into an updated case frame dictionary for the benefit of future researchers.

Further work is also needed to add support for instruments into the algorithm. This is extremely important as the most common elements in the “manner” frame in Becker’s algorithm deal with the manner in which a given instrument is used to perform the verb task.

Finally, one small problem is the way in which the data is printed to the screen. Any primitive action which performs multiple searches through the network will end up printing out separate lists for each search since they are not combined prior to being passed to Lisp’s format function. The output could be cleaned up fairly easily using a few list processing functions.

Appendix A: Case Frames and Semantics

Note: Some images and semantics are taken from Napieralski's Dictionary of CVA Case Frames at `http://www.cse.buffalo.edu/~rapaport/CVA/cvaresources.html`.

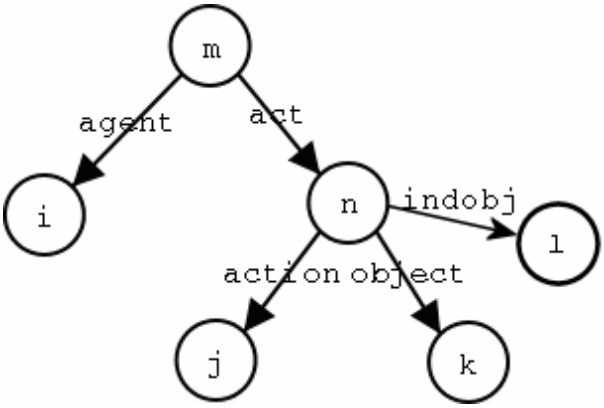

Figure 1: *Does*(i, *action-ditrans*(j, k, i)) frame

Semantics

[[m]] is the proposition that agent [[i]] performs action [[j]] with respect to object [[k]] and indirect object [[l]].

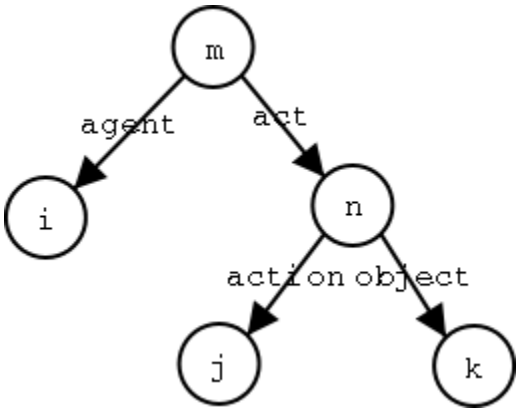

Figure 2: *Does*(i, *action-wrt*(j, k)) frame

Semantics

[[m]] is the proposition that agent [[i]] performs action [[j]] with respect to object [[k]].

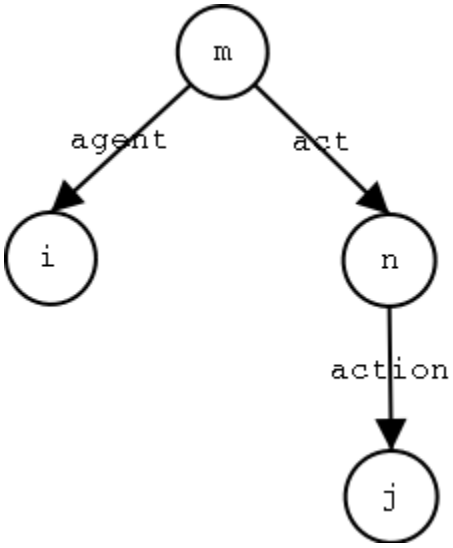

Figure 3: *Does*(i, *action-intrans*(j)) frame

Semantics

[[m]] is the proposition that agent [[i]] performs action [[j]].

Note: Due to SNeRE naming conventions, the *action* arc label has been changed to *action1*.

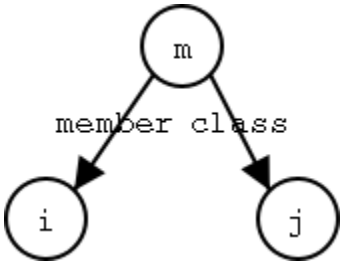

Figure 4: *Isa*(j, i) frame

Semantics

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

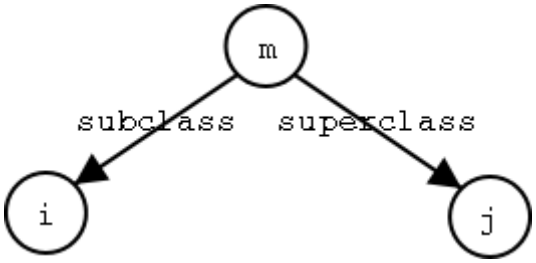

Figure 5: *Is-a-kind-of*(j, i) frame

Semantics

[[m]] is the proposition that [[i]] is a subclass of [[j]].

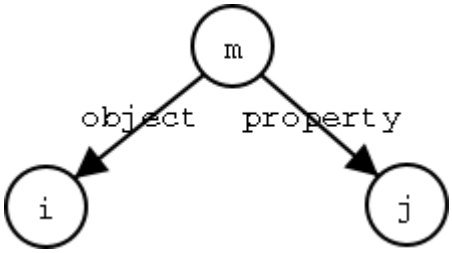

Figure 5: *Is*(j, i) frame

Semantics

[[m]] is the proposition that [[i]] has the property [[j]].

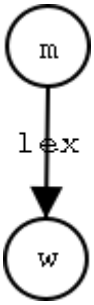

Figure 6: *thing-called*(w) frame

Semantics

[[m]] is the concept expressed by uttering [[w]].

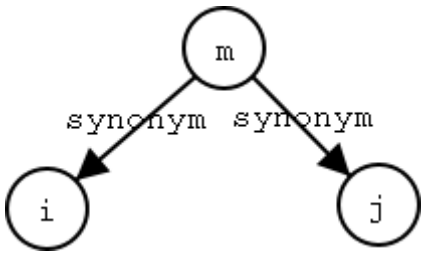

Figure 7: *are-synonymous(*i, j) frame

Semantics

[[m]] is the proposition that some concepts [[i]] and [[j]] are synonyms.

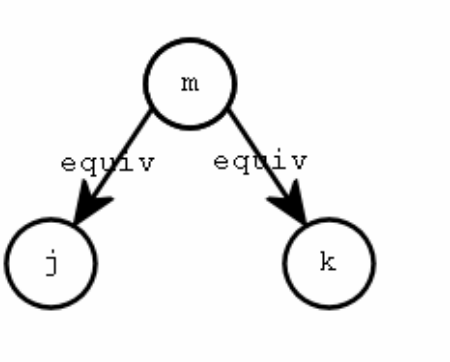

Figure 7: *Equivalent(*i, j) frame

Semantics

[[m]] is the proposition that some concepts [[i]] and [[j]] are equivalent.

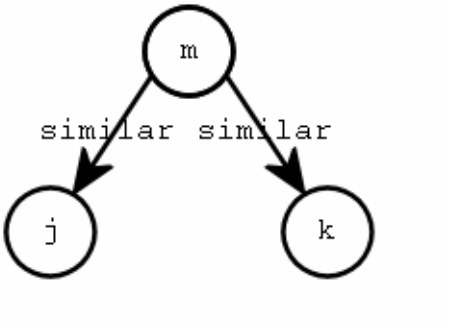

Figure 8: *Similar(*i, j) frame

Semantics

[[m]] is the proposition that some concepts [[i]] and [[j]] are equivalent.

Appendix B: Template for the SNeRE Verb Algorithm

```
;; ================================================
;; SNeRE Verb Algorithm, 1.0
;; by Joe Salazar - jsalazar@cse.buffalo.edu
;;
;; Future work:
;; Add support for instruments
;; Add support for the 'manner' frame
;; Change all instances of ‘Lex’ to ‘thing-called’ and
;; restructure the code layout
;;
;; ================================================

set-mode-3

;;define-frame Verb(class member)
define-frame Transitivity(verb-feature verb type)

define-frame CauseEffect(nil cause effect)

;; ================================================
;; Frames for classification.
;; Many of these are also used by the noun algorithm.
;; ================================================

define-frame Lex(nil lex)

define-frame Is-Named(nil object propername)
define-frame Is(nil property object)
define-frame Isa(nil class member)
define-frame Is-a-Kind-of(nil superclass subclass)

;; Equivalency relations.

define-frame Equivalent(nil equiv equiv)
define-frame Similar(nil similar similar)

define-frame are-synonymous(nil synonym synonym)

;; agent acting frames

define-frame Does(nil agent act)
define-frame action-intrans(nil action1)
define-frame action-wrt(nil action1 object1)
define-frame action-ditrans(nil action1 object1 indobject1)

;; ================================================
;; SNeRE action frames.
;; These frames define the complex and primitive actions
;; used by the algorithm.
;; ================================================

define-frame DefineVerb(action verb)
define-frame DetermineTransitivity(action verb)
define-frame DetermineSimilarActions(action verb)
```

```
define-frame DetermineDirectCause(action verb)
define-frame DetermineDirectEffect(action verb)
define-frame DetermineActionCause(action verb)
define-frame DetermineActionEffect(action verb)
define-frame DetermineAgentSuperclass(action verb)
define-frame DetermineObjectSuperclass(action verb)

define-frame FindVerb(action verb)
define-frame FindVerbMembership(action verb)
define-frame FindVerbTransitivity(action verb)
define-frame FindVerbProperties(action verb)
define-frame FindEquivalences(action verb)
define-frame FindSimilarActions(action object agent)
define-frame FindDirectCause(action verb agent)
define-frame FindDirectEffect(action verb agent)
define-frame FindActionCause(action verb agent1 agent2)
define-frame FindActionEffect(action verb agent1 agent2)
define-frame FindAgentSuperclass(action agent)
define-frame FindObjectSuperclass(action object)

define-frame Say(action object)

;define-frame snsequence2(action object1 object2)
;define-frame snsequence3(action object1 object2 object3)
;define-frame snsequence4(action object1 object2 object3 object4)

define-frame snsequence12(action object1 object2 snepsul::object3
           snepsul::object4 snepsul::object5 snepsul::object6
           snepsul::object7 snepsul::object8 snepsul::object9
           snepsul::object10 snepsul::object11 snepsul::object12)

;; =======================================================
;; Primitive action definitions.
;;
;; The primitive actions used by the algorithm are defined
;; and attached in this section.
;; =======================================================

^^
(define-primaction Say((object))
  (format t "~A~%" object))

(define-primaction FindVerb ((verb))
  "Print the name of the verb."
  (format t "Verb: ~A~%" verb))

(define-primaction FindVerbMembership ((verb))
  "Find all classes in which verb is a member."
  (format t "Class: ~A~%" (askwh (concatenate 'string "Isa(Lex(?x),Lex("
(princ-to-string verb) "))"))))

(define-primaction FindVerbTransitivity ((verb))
  "Print the transitivity of the verb."
  (format t "Transitivity: ~A~%" (askwh (concatenate 'string
"Transitivity(Lex(" (princ-to-string verb) "),?x)"))))
```

```
(define-primaction FindVerbProperties ((verb))
  "Find the properties of the verb."
  (format t "Properties: ~A~%" (askwh (concatenate 'string "Is(Lex(?x),Lex("
(princ-to-string verb) "))"))))

(define-primaction FindAgentSuperclass ((agent))
  "Find the properties of the verb."
  (format t "Agent: ~A~%" (askwh (concatenate 'string "Isa(Lex(?x),Lex("
(princ-to-string agent) "))"))))

(define-primaction FindObjectSuperclass ((object))
  "Find the properties of the verb."
  (format t "Object: ~A~%" (askwh (concatenate 'string "Isa(Lex(?x),Lex("
(princ-to-string object) "))"))))

(define-primaction FindEquivalences ((verb))
  "Find the direct equivalencies with the verb."
  (format t "Similar Actions: ~A ~A ~A~%"
(askwh (concatenate 'string
"Equivalent(Lex(" (princ-to-string verb) "),Lex(?x))"))
(askwh (concatenate 'string
"are-synonymous(Lex(" (princ-to-string verb) "),Lex(?x))"))
(askwh (concatenate 'string
"Similar(Lex(" (princ-to-string verb) "),Lex(?x))"))))

(define-primaction FindSimilarActions ((object) (agent))
  "Find all actions similar to the verb."
  (format t "Similar Actions: ~A~%" (askwh (concatenate 'string
  "Does(Lex(" (princ-to-string agent) "),
action-wrt(Lex(?x), Lex(" (princ-to-string object) ")))" ))))

(define-primaction FindDirectCause ((verb) (agent))
  "Find all cause relations with the verb."
  (format t "Cause: ~A~%" (askwh (concatenate 'string
"CauseEffect(Lex("(princ-to-string verb) "),
Does(Lex("(princ-to-string agent)"), action-intrans(Lex(?x)) ))"))))

(define-primaction FindDirectEffect ((verb) (agent))
  "Find all cause relations with the verb."
  (format t "Cause: ~A~%" (askwh (concatenate 'string
  "CauseEffect(Does(Lex(" (princ-to-string agent) "),
action-intrans(Lex(?x)) ),Lex("(princ-to-string verb) "))"))))

(define-primaction FindActionCause ((verb) (agent1) (agent2))
  "Find all action cause relations with the verb."
  (format t "Action Cause: ~A~%" (askwh (concatenate 'string
  "CauseEffect(Does(Lex(" (princ-to-string agent1) "),
action-intrans(Lex("(princ-to-string verb) "))),
Does(Lex("(princ-to-string agent2)"), action-intrans(Lex(?x)) ))"))))

(define-primaction FindActionEffect ((verb) (agent1) (agent2))
  "Find all action cause relations with the verb."
  (format t "Action Effect: ~A~%" (askwh (concatenate 'string
  "CauseEffect(Does(Lex("(princ-to-string agent2)"),
action-intrans(Lex(?x)) ), Does(Lex(" (princ-to-string agent1) "),
action-intrans(Lex(" (princ-to-string verb) "))))"))))
```

```
(attach-primaction do-one do-one do-all do-all believe believe
               disbelieve disbelieve adopt adopt unadopt unadopt
               snsequence12 snsequence sniterate sniterate
               snif snif withall withall withsome withsome
               ;; user defined primitive actions
               Say Say FindVerb FindVerb
               FindVerbProperties FindVerbProperties
               FindVerbMembership FindVerbMembership
               FindVerbTransitivity FindVerbTransitivity
               FindSimilarActions FindSimilarActions
               FindDirectCause FindDirectCause
               FindDirectEffect FindDirectEffect
               FindActionCause FindActionCause
               FindActionEffect FindActionEffect
               FindEquivalences FindEquivalences
               FindAgentSuperclass FindAgentSuperclass
               FindObjectSuperclass FindObjectSuperclass
)
^^


;; ========================================
;; Path definitions for "one or more" relations
;; ========================================

;; ========================================
;; Path definition for "superclass" argument.
;; ========================================

define-path superclass (compose superclass (kstar (compose subclass-
 ! superclass)))

;; ========================================
;; Path definition for "superordinate" argument.
;; ========================================

define-path class (compose class (kstar (compose subclass- ! superclass)))

;; ========================================
;; Path definition for equivalency chains.
;; ========================================

define-path equiv (compose equiv (kstar (compose equiv- ! equiv)))

;; ========================================
;; Path definition for synonym chains.
;; ========================================

define-path synonym (compose synonym (kstar (compose synonym- ! synonym)))

;; ========================================
;; Path definition for similarity chains.
;; ========================================

define-path similar (compose similar (kstar (compose similar- ! similar)))
```

```
;; ========================================
;; Now add the representation of the sentence.
;; ========================================



;; ============================================================
;; Rules for determining superclasses of agents and objects.
;;
;; The action frame is checked in order to determine the
;; agent or object as appropriate. If either are present then
;; they are printed in the verb frame.
;; ============================================================

all(v)(Isa(Lex(verb), Lex(v)) => ActPlan(DetermineAgentSuperclass(v),
       withall(?x, Does(Lex(?x), action-intrans(Lex(v))),
       FindAgentSuperclass(x)))).

all(v)(Isa(Lex(verb), Lex(v)) => ActPlan(DetermineObjectSuperclass(v),
       withall({?x, ?y}, Does(Lex(?x), action-wrt(Lex(v), Lex(?y))),
       FindObjectSuperclass(y)))).

;; ============================================================
;; Rules for determining transitivity.
;;
;; Checks the action frame of the verb to see if there's an
;; indirect object in use. If so, then the verb is ditransitive.
;; If not, then it checks the action frame to see if the verb
;; is done with respect to an object. If so, then the verb is
;; transitive. If not, then the verb must be intransitive.
;; ============================================================

all(v)(Isa(Lex(verb), Lex(v)) => ActPlan(DetermineTransitivity(v),
        withall({?x, ?y, ?z}, Does(?x, action-ditrans(Lex(v), ?y, ?z)),
        believe(Transitivity(Lex(v), ditransitive)),
            withall({?x, ?y}, Does(?x, action-wrt(Lex(v), ?y)),
                  believe(Transitivity(Lex(v), transitive)),
                  believe(Transitivity(Lex(v), intransitive)))))).

;; ====================================================================
;; Rules for determining similar actions.
;;
;; Checks the action frame to see if any equivalent, synonymous or
;; similar actions exist in the network. These actions are represented
;; by using the equiv, synonym and similar frames respectively. If
;; these are not used with respect to the verb then the program scans
;; the network for other actions which could be considered similar.
;; ====================================================================

;; Ensuring that the equivalence relationships trigger by deriving
;; the reverse of every frame.

all(x, y)(Equivalent(x, y) => Equivalent(y, x)).
all(x, y)(are-synonymous(x, y) => are-synonymous(y, x)).
all(x, y)(Similar(x, y) => Similar(y, x)).
```

```
all(v)(Isa(Lex(verb), Lex(v)) => ActPlan(DetermineSimilarActions(v),
       withall(?x, are-synonymous(Lex(v), Lex(?x)), FindEquivalences(v),
       withall(?y, Equivalent(Lex(v), Lex(?y)), FindEquivalences(v),
       withall(?z, Similar(Lex(v), Lex(?z)), FindEquivalences(v),
       withall({?t,?w}, Does(?t, action-wrt(Lex(v), Lex(?w))),
       withall({?a,?b}, Does(Lex(?a), action-wrt(Lex(?b), Lex(w))),
       FindSimilarActions(w, a) ))))))).

;; ============================================================
;; Cause and effect frame rules.
;;
;; There are two types of cause and effect relations, direct and
;; action-related. Each is checked individually to determine if
;; they exist. If they do, then the relevant rule is triggered.
;;
;; ============================================================

all(v)(Isa(Lex(verb), Lex(v)) => ActPlan(DetermineDirectCause(v),
       withall({?x,?y}, CauseEffect(Lex(v), Does(Lex(?x),
       action-intrans(?y))), FindDirectCause(v, x)))).

all(v)(Isa(Lex(verb), Lex(v)) => ActPlan(DetermineDirectEffect(v),
       withall({?x,?y}, CauseEffect(Does(Lex(?x), action-intrans(?y)),
       Lex(v)), FindDirectEffect(v, x)))).

all(v)(Isa(Lex(verb), Lex(v)) => ActPlan(DetermineActionCause(v),
       withsome({?x,?y,?z}, CauseEffect(Does(Lex(?x),
       action-intrans(Lex(v))), Does(Lex(?y),
       action-intrans(Lex(?z)))), FindActionCause(v, x, y)))).

all(v)(Isa(Lex(verb), Lex(v)) => ActPlan(DetermineActionEffect(v),
       withsome({?x,?y,?z}, CauseEffect(Does(Lex(?x),
       action-intrans(Lex(?y))), Does(Lex(?z),
       action-intrans(Lex(v)))), FindActionEffect(v, z, x)))).

;; ============================================================
;; Algorithm flow of control.
;;
;; All of the complex actions and primitive actions involved in
;; defining the verb frame are called in sequential order given
;; below.
;; ============================================================

all(v)(Isa(Lex(verb), Lex(v)) => ActPlan(DefineVerb(v),
      snsequence12(DetermineTransitivity(v), FindVerb(v),
                FindVerbMembership(v), FindVerbTransitivity(v),
                FindVerbProperties(v), DetermineDirectCause(v),
                DetermineDirectEffect(v), DetermineActionCause(v),
                DetermineActionEffect(v), DetermineSimilarActions(v),
                DetermineAgentSuperclass(v), DetermineObjectSuperclass(v)))).

;; ======================
;; Define the verb.
;; ======================

perform DefineVerb(perambulate)
```

References

Becker, Chris (2005), “Contextual Vocabulary Acquisition of Verbs”

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.

Rapaport, William J., & Kibby, Michael W. (2002), "ROLE: Contextual Vocabulary Acquisition: From Algorithm to Curriculum".

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.

Shapiro, Stuart C. (1989), "The CASSIE Projects: An Approach to Natural Language Competence", in João P. Martins & Ernesto M. Morgado (eds.), *EPIA 89: 4th Portugese Conference on Artificial Intelligence Proceedings*, Lecture Notes in Artificial Intelligence 390 (Berlin: Springer-Verlag): 362-380.