Verb Acquisition in Computational Cognitive Agents

Rajeev Sood

May 6, 2002

CSE663

Abstract

Learning the meaning of a word from its context is a complex problem for a cognitive agent to solve. It is not an easy task for a computational cognitive agent in particular. By context, I mean the surrounding text as well as the grammatical structure and background knowledge associated with the unknown word. As part of an interdisciplinary project, an algorithm has been proposed to allow a cognitive agent to learn the meaning of a word from its context (Rapaport 2000). This paper deals with the way in which the meaning of a verb can be derived from context.

1 Introduction

Learning the meaning of a word from its context is a complex problem for a cognitive agent to solve. It is not an easy task for a computational cognitive agent in particular. By context, I mean the surrounding text as well as the grammatical structure and background

knowledge associated with the unknown word. As part of an interdisciplinary project, an algorithm has been proposed to allow a cognitive agent to learn the meaning of a word from its context (Rapaport 2000). This algorithm has a different process it follows depending on the function of the word in the sentence.

Currently, there are two different algorithms: one for nouns, and one for verbs. The focus of this paper will be the algorithm devised for verbs. This algorithm was originally written by Karen Ehrlich (see reference list for Ehrlich's 1995 paper). The purpose of this paper is to provide an in-depth analysis of this algorithm and suggest improvements to it.

2 Case Frame Dictionary for Erlich's Verb Algorithm

Ehrlich's algorithm works on the premise that the SNePS representation of the passage uses only the case frames described in this section. Any other information represented using any other case frame is ignored by the algorithm.

object/property (Broklawski 2002)

Syntax:

(assert object i property j)

Semantics:

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

Sample Context:

Jack is heavy.

```
(describe (assert object #Jack proper-name (build lex "Jack")))
(describe (assert object *Jack property (build lex "heavy")))
```

member/class (Broklawski 2002)

Syntax:

(assert member i class j)

Semantics:

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

Sample Context:

Evelyn is a person.

```
(describe (assert object #Eve proper-name (build lex "Evelyn")))
```

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

Required Usage:

If the class in question is basic-level (e.g., table, person) then you *must* use the member/class case frame, and *not* the object1/rel/object2 case frame.

members/class

Syntax:

(assert members i class j)

Semantics:

[[m]] is the proposition that [[i]] are (members of class) [[j]].

Sample Context:

Knights are people.

(describe (assert members (build lex "Knights") class (build lex "people")))

Required Usage:

If the class in question is basic-level (e.g., tables, people) then you *must* use the members/class case frame, and *not* the objects1/rel/object2 case frame.

agent/act/object/indobj

Syntax:

(build agent i act j object k indobj l)

Semantics:

[[m]] is the proposition that agent [[i]] performs act [[j]] upon [[k]] with repect to [[l]].

Sample Context:

Joe hits the ball to John.

(describe (assert object #Joe proper-name (build lex "Joe")))
(describe (assert object #John proper-name (build lex "John")))
(describe (assert member #Ball class (build lex "ball")))
(describe (assert agent *Joe act (build lex "hit") object *Ball indobj *John))

agent/act/object (Broklawski 2002)

Syntax:

(build agent i act j object k)

Semantics:

[[m]] is the proposition that agent [[i]] performs act [[j]] with repect to [[k]].

Sample Context:

Joe hits the ball.

(describe (assert object #Joe proper-name (build lex "Joe")))
(describe (assert member #Ball class (build lex "ball")))
(describe (assert agent *Joe act (build lex "hit") object *Ball))

agent/act (Broklawski 2002)

Syntax:

(build agent i act j)

Semantics:

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

Sample Context:

Joe sleeps.
(describe (assert object #Joe proper-name (build lex "Joe")))
(describe (assert agent *Joe act (build lex "sleep")))

object1/rel/object2 (Broklawski 2002)

Syntax:

(assert object1 i rel j object2 k)

Semantics:

[[m]] is the proposition that [[j]]([[i]], [[k]]).

Sample Context:

Jack is next to Fred.
(describe (assert object #Jack proper-name (build lex "Jack")))
(describe (assert object #Fred proper-name (build lex "Fred")))
(describe (assert object1 *Jack rel (build lex "next to") object2 *Fred))

Special Usage (class inclusion):

rel is "ISA"

If the class in question is either a subordinate (e.g., coffee table) or a superordinate (e.g., furniture) then the object1/rel/object2 case frame *must* be used, where object1 points to some individual, rel points to ISA, and object2 points to the class in question.

rel is "enable"

This indicates that the first object enables the second object. In other words, by existing, the first object allows for the possibility of the second one.

objects1/rel/object2 (rel is "ARE") (Broklawski 2002)

Syntax:

(assert objects1 i rel j object2 k)

Semantics:

[[m]] is the proposition that [[j]]([[i]], [[k]]).

Sample Context:

Knights varled their shields.
(describe (assert agent #KS act (build lex "varl") object #SH time #KVS))
(describe (assert objects1 *KS rel (build lex "are") object2 (build lex "knight")))
(describe (assert members *SH class (build lex "shield")))

Special Usage:

If the class in question is either a subordinate (e.g., coffee table) or a superordinate (e.g., furniture) then the objects1/rel/object2 case frame *must* be used, where objects1 points to some definite plural entity (collection), rel points to ARE, and object2 points to the class in question.

mode/object (mode is "presumably") (Broklawski 2002)

Syntax:

(build mode i object j)

Semantics:

[[m]] is the proposition that the modal concept [[i]] is applied to the object [[j]].

Sample Context:

Presumably, Joe is smart.
(describe (assert object #Joe proper-name (build lex "Joe")))
(describe (assert mode (build lex "presumably") object (build object *Joe property
(build lex "smart"))))

ant/cq

Syntax:

(assert ant i cq j)

Semantics:

[[r]] is the rule that if structured individual [[i]] is true, then so is [[j]].

Sample Context:

For all v1, if v1 is a student, then v1 is a person.
(describe (assert forall $v1
ant (build member *v1 class student)
cq (build member *v1 class person)))

&ant/cq

Syntax:

(assert &ant h &ant i cq j)

Semantics:

[[r]] is the rule that if structured individuals [[h]] and [[i]] are true, then so is [[j]].

Sample Context:

For all v1, if v1 is a student and v1 is a person, then v1 is a mammal.
(describe (assert forall $v1
&ant (build member *v1 class student)
&ant (build member *v1 class person)
cq (build member *v1 class mammal)))

cause/effect

Syntax:

(assert cause i effect j)

Semantics:

[[m]] is the proposition that structured individual [[i]] enables structured individual [[j]].

Sample Context:

Joe broke the chair because he fell.
(describe (assert object #Joe proper-name (build lex "Joe")))
(describe (assert cause (assert agent *Joe act (build lex "fall"))
effect (assert agent *Joe act (build lex "break") object (build lex
"chair"))))

3 Algorithm Analysis

4.1 Overview

This algorithm is designed to produce a definition of a verb from context dynamically. In other words, as new information is added to the knowledge base about the verb, including background knowledge and new occurrences of the verb in question, the definition of the verb changes.

Both Karen Ehrlich's original algorithm and my improved version of her algorithm define a verb using five basic elements: the subject, the object (if it exists), the indirect object (if it exists), the things that the verb enables, and the things that enable the verb. Note that the search paths for all of these are in the appendix, section 8.4 of this document.

The first element, the subject, produces a result by proceeding in the following manner: If the subject is neither a base-level category or an animal, return any category information available. Otherwise if the subject is not a base-level category, return the animal category, otherwise return the base-level category. The second element, the object, and the third element, the indirect object, both of which may or may not exist, follow the same procedure as the first element. The fourth and fifth elements follow a more complex decision process which is explained best in diagrams. Refer to section 8.4 of this document listed under cause (for fourth) and effect (for fifth). The major difference between these two algorithms is in how this information is put together.

4.2 Karen Ehrlich's Original Algorithm

The original algorithm is designed to rely upon the assumption that the verb being defined has been labeled as being bitransitive, transitive, reflexive, or intransitive. Bitransitive means that the verb being defined has both a direct and indirect object. Transitive implies that

the verb has only a direct object. Reflexive is a transitive form, but one in which the subject performs an action upon itself. Intransitive means that a verb has neither a direct or indirect object.

Keeping this in mind, Karen Ehrlich's algorithm follows these steps:

1. Categorize the subject and report it.
2. Determine the form of the verb:
    a. If bitransitive, categorize the direct and indirect objects and report them.
    b. If transitive, categorize the direct object and report it.
    c. If reflexive, report that the subject performs that act upon itself.
    d. If intransitive, go to the next step.
3. Find anything that the verb occurring enables (cause function) and report it.
4. Find anything that occurs that enables the verb to occur (effect function) and report it.

As can be seen, this algorithm is dependant upon the type of verb being specified. This is a limiting factor of the algorithm. Another limiting factor of the algorithm, is that in a number of places, the algorithm will search to see if something exists, and then will search again to report the information if it does exist. Another limitation is that in returning information from the cause function, the algorithm will return only information from antecedant (ant) arcs if they exist, and ignore information from &ant (when the antecedant is one of many) arcs which may or may not exist.

4.3 My Improved Version

In my version of this algorithm, I addressed the issues that I mentioned in the previous section. First of all, my algorithm reports back the direct and indirect object information if it exists, and does not need a type specified. It does not, however, distinguish between a transitive and reflexive verb. I also fixed the problem of doing double searches by setting a variable at the beginning of the function, then using that variable afterwards instead of searching again. I did not, however, use local variables (let), but instead used global variables (setq). I furthermore fixed the limitation that prevented the cause function from returning information from both ant and &ant arcs, if it exists.

The algorithm works as follows:

1. Categorize the subject and report it.
2. If it exists, categorize the direct object and report it.
3. If it exists, categorize the indirect object and report it.
4. Find anything that the verb occurring enables (cause function) and report it.
5. Find anything that occurs that enables the verb to occur (effect function) and report it.

I have also implemented a basic trace function. It returns a list of the functions that contributed information to the definition.

5 Sample Runs

In the appendix, section 8.3, a sample run is provided. The first section is the demo file. Then, section 8.3.1 is the demo run using the original algorithm. Section 8.3.2 is the run using

the new algorithm. The demo provides seven samples in it. All of these samples are tested on the verb throw (Demo courtesy of Justin Del Vecchio).

In the first sample, the sentence is transitive, but the fact that it is transitive is not represented. Note that the original algorithm only reports for the "default case", which is the intransitive case. The new version, however, picks up the fact that the sentence is transitive and reports the direct object as well.

In the second sample, the fact that the verb is transitive is represented. In this instance, both algorithms behave the same as each other. The new algorithm reports as much information as the first sample, when the transitive form wasn't specified. The third and fourth samples behave similarly, except that sample three specifies the verb is transitive when it really is bitransitive, while the fourth specifies the type correctly. In both of these instances, the new algorithm correctly reports the same information. The old algorithm reports differently, at first not finding the indirect object, but reporting all the information correctly the second time.

Sample five specifies that throw is bitransitive, but offers both transitive and bitransitive examples. Both algorithms return the same information. They both treat throw as being bitransitive in all instances, and makes no distinction between a bitransitive instance and a transitive one. Sample six returns the same information as sample five, but is represented as being both bitransitive and transitive. Both algorithms report slightly incorrect information in these samples by not making separate reports for each occurrence of the verb. Sample seven demonstrates the ability of the algorithms to gather information from rules in the knowledge base about the verb.

6 Future Work

The work that still needs to be done can be divided into two categories. The first is immediate next steps. The first thing to be done is to make all global variables local by using let statements. The second thing to be done is to figure out a way to include the reflexive case in the new version of the algorithm. Third, we need to devise a way to return each occurrence of the verb separately instead of jumbled together.

The second category is long-term future work. First is the implementation of a parser and language generator. Second would be to develop a concept of basic level categories that a verb can fit into. Third, build into the algorithm a way of keeping track of the states that the known part of the world is in prior to the unknown verb occurring and immediately following the action of the verb. By tracking what changes are taking place during the time that the verb occurs, a better definition can be derived.

7 References

Broklawski, Marc K.; Rapaport, William J; & Ehrlich, Karen(2002), "A Dictionary of CVA SNePS Case Frames", http://www.cse.buffalo.edu/~mkb3/case.frame.dictionary/case.frame.index.html.

Ehrlich, Karen (1995), "Automatic Vocabulary Expansion through Narrative Context", *Technical Report 95-09* (Buffalo: SUNY Buffalo Department of Computer Science).

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.

8 Appendices

8.1 Karen Ehrlich's Original Verb Algorithm

```
(in-package :snepsul)

;-------------------------------------------------------------------------------
;
;               function: defn_verb
;               input:          a verb to be defined
;               output: predicate structure of , with categorization of arguments;
;                               causal/enablement information; eventually to include primitive
;                               act of which  is a type (if any).
;               calls:  report_bitransitive, or report_transitive, or report_reflexive,
;                               or report_intransitve, as appropriate.
;               NOTE: #3! is a macro which allows snepsul commands to be invoked from
;                    within lisp functions; obviates need for references to pkgs, etc.
;-------------------------------------------------------------------------------
(defun defn_verb (verb)
    (setq czs (cause verb))
    (setq efs (car (effect verb)))
    (cond (#3! ((deduce property (build lex "bitransitive")
                                        object (build lex ~verb)))
                 (report_bitransitive verb czs efs))
                (#3! ((deduce property (build lex "transitive")
                                        object (build lex ~verb)))
                 (report_transitive verb czs efs))
                (#3! ((deduce property (build lex "reflexive")
                                        object (build lex ~verb)))
                 (report_reflexive verb czs efs))
                (t (report_intransitive verb czs efs))))


;-------------------------------------------------------------------------------
;
;               function: report_bitransitive
;               input: a verb to be defined
;               output: predicate structure of , with categorization of arguments;
;                               causal/enablement information; eventually to include primitive
;                               act of which  is a type (if any).
;               calls:          categorize_subject, categorize_object, categorize_indobject,
;                               cause, effect, (prim_base currently undefined)
;-------------------------------------------------------------------------------
(defun report_bitransitive (verb czs efs)
    (list 'a
        (categorize_subject verb)
                'can verb 'a
        (categorize_object verb)
                'to 'a
```

```
        (categorize_indobject verb)
                'result= #3!((describe ~czs))
                'enabled 'by= #3!((describe ~efs))
;               (prim_base verb)
))


;-------------------------------------------------------------------------
;
;               function: report_transitive
;               input:  a verb to be defined
;               output: predicate structure of , with categorization of arguments;
;                               causal/enablement information; eventually to include primitive
;                               act of which  is a type (if any).
;               calls:          categorize_subject, categorize_object, cause, effect,
;                               (prim_base currently undefined)
;-------------------------------------------------------------------------
(defun report_transitive (verb czs efs)
    (list 'a
                (categorize_subject verb)
                'can verb 'a
        (categorize_object verb)

                'result= czs
                'enabled 'by= #3!((describe ~efs))
;               (prim_base verb)
))


;-------------------------------------------------------------------------
;
;               function: report_reflexive
;               input:  a verb to be defined
;               output: predicate structure of , with categorization of arguments;
;                               causal/enablement information; eventually to include primitive
;                               act of which  is a type (if any).
;               calls:          categorize_subject, cause, effect,
;                               (prim_base currently undefined)
;-------------------------------------------------------------------------
(defun report_reflexive (verb czs efs)
    (list 'a
                (categorize_subject verb)
        'can verb 'itself
                'result= #3!((describe ~czs))
                'enabled 'by= #3!((describe ~efs))
;               (prim_base verb)
))



;-------------------------------------------------------------------------
;
;               function: report_intransitive
;               input:  a verb to be defined
;               output: predicate structure of , with categorization of argument;
;                               causal/enablement information; eventually to include primitive
```

```
;                       act of which  is a type (if any).
;           calls:      categorize_subject, cause, effect,
;                       (prim_base currently undefined)
;-------------------------------------------------------------------------
(defun report_intransitive (verb czs efs)
   (list 'a
            (categorize_subject verb)
            'can verb
            'result= #3!((describe ~czs))
            'enabled 'by= #3!((describe ~efs))
;           'result= czs
;           'enabled 'by= efs
;           (prim_base verb)
))


;-------------------------------------------------------------------------
;
;           function: categorize_subject
;           input:  a verb to be defined
;           output: categorization of encountered subjects of  as (in order
;                       of preference) 1)belonging to some basic level category,
;                       2)belonging to some subclass of animal, or 3)belonging to some
;                       miscellaneous (but known) class.
;           calls:      base_cat_subj, anim_subj, some_cat_subj, emptyp
;-------------------------------------------------------------------------
(defun categorize_subject (verb)
   (cond ((emptyp (list (base_cat_subj verb)
                        (anim_subj verb)))
            (setq subject (some_cat_subj verb))
            (if (listp subject)
                        (cathelper subject nil)
                        subject))
         ((emptyp (base_cat_subj verb))
          (cathelper (anim_subj verb) nil))
         (t (cathelper (base_cat_subj verb) nil))))
;-------------------------------------------------------------------------
;
;           function: person_subj  (function not currently used in defining verb)
;           input:  a verb to be defined
;           output: the atom 'person, if a member of the class person has been
;                       encountered as the subject of
;-------------------------------------------------------------------------
(defun person_subj (verb)
   (cond ((AND #3! ((deduce agent $vsub act (build lex ~verb)))
                        #3! ((deduce member *vsub class (build lex "person"))))
            'person)))
;-------------------------------------------------------------------------
;
;           function: anim_subj
;           input:  a verb to be defined
;           output: a list of the kinds of animal which have been known to
;-------------------------------------------------------------------------
(defun anim_subj (verb)
   (cond ((AND #3! ((deduce agent $vsub1 act (build lex ~verb)))
                        #3! ((deduce member *vsub1 class (build lex "animal"))))
```

```
          (append #3! ((find (compose lex- class- ! member
                                agent- ! act lex) ~verb))
                      #3! ((find (compose lex- class- ! members
                                agent- ! act lex) ~verb))))
;         (#3! ((find (compose lex- class-  member
;                             agent-  act lex) ~verb))
;          (list #3! ((find (compose lex- class-  member
;                              agent-  act lex) ~verb))))
))


;-------------------------------------------------------------------------
;
;               function: base_cat_subj
;               input:  a verb to be defined
;               output: a list of basic level categs. which have been known to
;-------------------------------------------------------------------------
(defun base_cat_subj (verb)
    (cond ((AND #3! ((deduce agent $vsub2 act (build lex ~verb)))
                                (OR #3! ((deduce member *vsub2 class $vbasic))
                                   #3! ((deduce members *vsub2 class $vbasic)))
                                #3! ((deduce object1 *vbasic rel "ISA"
                          object2 (build lex "basic ctgy"))))
          (append #3! ((find (compose lex- class- ! member
                                agent- ! act lex) ~verb))
                      #3! ((find (compose lex- class- ! members
                                agent- ! act lex) ~verb))))))
;-------------------------------------------------------------------------
;
;               function: some_cat_subj
;               input:  a verb to be defined
;               output: a list of the kinds of things which have been known to
;                             or the atom 'something, if nothing known about what can .
;-------------------------------------------------------------------------
(defun some_cat_subj (verb)
    (cond ((AND #3! ((deduce agent $vsub3 act (build lex ~verb)))
                        (OR #3! ((deduce object1 *vsub3 rel "ISA" object2 $somecat))
                           #3! ((deduce objects1 *vsub3 rel "ARE" object2 $somecat))))
          (append #3! ((find (compose lex- object2- ! object1
                                                        agent- ! act lex) ~verb))
                      #3! ((find (compose lex- object2- ! objects1
                                                        agent- ! act lex) ~verb))))
;         (#3! ((find (compose lex- object2-  object1
;                             agent-  act lex) ~verb))
;          (list #3! ((find (compose lex- object2-  object1
;                                                        agent-  act lex) ~verb))))
          (t 'something)))


;-------------------------------------------------------------------------
;
;               function: categorize_object
;               input:  a verb to be defined
;               output: categorization of encountered objects of  as (in order
;                             of preference) 1)belonging to some basic level category,
;                             2)belonging to some subclass of animal, or 3)belonging to some
```

```
;                       miscellaneous (but known) class.
;           calls:      base_cat_obj, anim_obj, some_cat_obj, emptyp
;-------------------------------------------------------------------------
(defun categorize_object (verb)
   (cond ((emptyp (list (base_cat_obj verb)
                        (anim_obj verb)))
          (setq object (some_cat_obj verb))
          (if (listp object)
                        (cathelper object nil)
                        object))
         ((emptyp (base_cat_obj verb))
          (cathelper (anim_obj verb) nil))
         (t (cathelper (base_cat_obj verb) nil))))
;-------------------------------------------------------------------------
;
;           function: person_obj  (function not currently used in defining verb)
;           input:  a verb to be defined
;           output: the atom 'person, if a member of the class person has been
;                       encountered as the object of
;-------------------------------------------------------------------------
(defun person_obj (verb)
   (cond ((AND #3! ((deduce object $vobj agent $vsub4 act (build lex ~verb)))
                        #3! ((deduce member *vobj class (build lex "person"))))
          'person)))


;-------------------------------------------------------------------------
;
;           function: base_cat_obj
;           input:  a verb to be defined
;           output: a list of basic categs. of things known to have been ed.
;-------------------------------------------------------------------------
(defun base_cat_obj (verb)
   (cond ((AND #3! ((deduce object $vobj1 agent $vsub5 act (build lex ~verb)))
                        #3! ((deduce member *vobj1 class $vbasic1))
                        #3! ((deduce object1 *vbasic1 rel "ISA"
                                        object2 (build lex "basic ctgy"))))
         (append #3! ((find (compose lex- class- ! member
                                                object- ! act lex) ~verb))
                 #3! ((find (compose lex- class- ! members
                                                object- ! act lex) ~verb))))
;         (#3! ((find (compose lex- class-  member
;                                                object-  act lex) ~verb))
;          (list #3! ((find (compose lex- class-  member
;                                                object-  act lex) ~verb))))
))
;-------------------------------------------------------------------------
;
;           function: anim_obj
;           input:  a verb to be defined
;           output: a list of the kinds of animals known to have been ed.
;-------------------------------------------------------------------------
(defun anim_obj (verb)
   (cond ((AND #3! ((deduce agent $vsub6 object $vobj2 act (build lex ~verb)))
                        #3! ((deduce member *vobj2 class (build lex "animal"))))
         (append #3! ((find (compose lex- class- ! member
```

```
                                                    object- ! act lex) ~verb))
              #3! ((find (compose lex- class- ! members
                                                    object- ! act lex) ~verb))))))


;-------------------------------------------------------------------------
;
;             function: some_cat_obj
;             input:  a verb to be defined
;             output: a list of the kinds of things known to have been ed,
;                           or the atom 'something, if nothing known about what can
;                           be ed.
;-------------------------------------------------------------------------

(defun some_cat_obj (verb)
   (cond ((AND #3! ((deduce agent $vsub7 object $vobj3 act (build lex ~verb)))
                  (OR #3! ((deduce object1 *vobj3 rel "ISA" object2 *somecat))
                     #3! ((deduce object1 *vobj3 rel "ISA" object2 *somecat))))
         (append #3! ((find (compose lex- object2- ! object1
                                                    agent- ! act lex) ~verb))
              #3! ((find (compose lex- object2- ! objects1
                                                    agent- ! act lex) ~verb))))
;         (#3! ((find (compose lex- object2-  object1
;                                                    agent-  act lex) ~verb))
;          (list #3! ((find (compose lex- object2-  object1
;                                                    agent-  act lex) ~verb))))
               (t 'something)))



;-------------------------------------------------------------------------
;
;             function: categorize_indobject
;             input:  a verb to be defined
;             output: categorization of encountered indirect objects of  as
;                           (in order of preference) 1)belonging to some basic level
;                           category, 2)belonging to some subclass of animal, or
;                           3)belonging to some miscellaneous (but known) class.
;             calls:        base_cat_indobj, anim_indobj, some_cat_indobj, emptyp
;-------------------------------------------------------------------------
(defun categorize_indobject (verb)
   (cond ((emptyp (list (base_cat_indobj verb)
                           (anim_indobj verb)))
               (setq indobject (some_cat_indobj verb))
               (if (listp indobject)
                           (cathelper indobject nil)
                           indobject))
        ((emptyp (base_cat_indobj verb))
         (cathelper (anim_indobj verb) nil))
        (t (cathelper (base_cat_indobj verb) nil))))
;-------------------------------------------------------------------------
;
;             function: person_indobj  (function not currently used in defining verb )
;             input:  a verb to be defined
;             output: the atom 'person, if a member of the class person has been
;                           encountered as the indirect object of
```

```
;-------------------------------------------------------------------------
(defun person_indobj (verb)
    (cond ((AND #3! ((deduce indobj $vindobj object $vobj4
                                        agent $vsub8 act (build lex ~verb)))
                    #3! ((deduce member *vindobj class (build lex "person"))))
                'person)))


;-------------------------------------------------------------------------
;
;               function: base_cat_indobj
;               input:  a verb to be defined
;               output: a list of basic categs. of things encountered as indirect
;                       object of
;-------------------------------------------------------------------------
(defun base_cat_indobj (verb)
    (cond ((AND #3! ((deduce object $vobj5 agent $vsub9
                                        indobj $vindobj1 act (build lex ~verb)))
                    #3! ((deduce member *vindobj1 class $vbasic1))
                    #3! ((deduce object1 *vbasic1 rel "ISA"
                object2 (build lex "basic ctgy"))))
          (append #3! ((find (compose lex- class- ! member
                                                indobj- ! act lex) ~verb))
                #3! ((find (compose lex- class- ! members
                                                indobj- ! act lex) ~verb))))
;         (#3! ((find (compose lex- class- member
;                                               indobj- act lex) ~verb))
;          (list #3! ((find (compose lex- class- member
;                                               indobj- act lex) ~verb))))
))
;-------------------------------------------------------------------------
;
;               function: anim_indobj
;               input:  a verb to be defined
;               output: a list of the kinds of animals known to have been the indirect
;                       object of .
;-------------------------------------------------------------------------
(defun anim_indobj (verb)
    (cond ((AND #3! ((deduce agent $vsub9 object $vobj6
                                        indobj $vindobj2 act (build lex ~verb)))
                    #3! ((deduce member *vindobj2 class (build lex "animal"))))
          (append #3! ((find (compose lex- class- ! member
                                                indobj- ! act lex) ~verb))
                #3! ((find (compose lex- class- ! members
                                                indobj- ! act lex) ~verb))))
))


;-------------------------------------------------------------------------
;
;               function: some_cat_indobj
;               input:  a verb to be defined
;               output: a list of the kinds of things known to have been the indirect
;                       object of , if any;
;                       the atom 'something if nothing known about what can be the
;                       indirect object of
```

```
;-------------------------------------------------------------------------
(defun some_cat_indobj (verb)
   (cond ((AND #3! ((deduce agent $vsub10 object $vobj7
                                        indobj $vindobj3 act (build lex ~verb)))
                   (OR #3! ((deduce object1 *vindobj3 rel "ISA" object2 *somecat))
                      #3! ((deduce object1 *vindobj3 rel "ISA" object2 *somecat))))
         (append #3! ((find (compose lex- object2- ! object1
                                                  indobj- ! act lex) ~verb))
               #3! ((find (compose lex- object2- ! objects1
                                                  indobj- ! act lex) ~verb))))
;          (#3! ((find (compose lex- object2- object1
;                                                  indobj- act lex) ~verb))
;          (list #3! ((find (compose lex- object2- object1
;                                                  indobj- act lex) ~verb))))
                (t 'something)))


;-------------------------------------------------------------------------
;
;              function: cause
;              input:        a verb to be defined
;              output: a list containing the result of .  List will contain
;              either propositions (molecular nodes) or patterns for them,
;              from rules (pattern node).
;-------------------------------------------------------------------------
(defun cause (verb)
   (cond (#3! ((deduce object1 (build agent *vsub11 act (build lex ~verb))
                                    rel (build lex "enable")
                                    object2 $goal))
         (cond (#3! ((find (compose cq- ! ant act lex) ~verb))
                  (list #3! ((find (compose cq- ! ant act lex) ~verb))
                        'or 'to 'enable
                         #3! ((find (compose lex- act- object2- ! rel lex) "enable"
                                    (compose lex- act- object2- ! object1 act lex)
                                    ~verb))))
                     (t (list 'to 'enable
                        #3! ((find (compose lex- act- object2- ! rel lex)
                                          "enable"
                                    (compose lex- act- object2- ! object1 act lex)
                                    ~verb))))))

              (#3! ((deduce mode (build lex "presumably")
                                    object (build object1 (build agent *vsub11
                                                                  act (build lex ~verb)
                                                                  time $vtime)
                                                rel (build lex "enable")
                                                object2 $goal)))
         (cond (#3! ((find (compose cq- ! ant act lex) ~verb))
                  (list #3! ((find (compose cq- ! ant act lex) ~verb))
                        'or 'to 'enable
                              #3! ((find (compose lex- act- object2- rel lex) "enable"
                                 (compose lex- act- object2- object1 act lex) ~verb
                                 (compose lex- act- object2- object- ! mode lex)
                                 "presumably"))))
                  (t (list 'to 'enable
                           #3! ((find (compose lex- act- object2- rel lex) "enable"
```

```
                                        (compose lex- act- object2- object1 act lex) ~verb
                                        (compose lex- act- object2- object- ! mode lex)
                                        "presumably"))))))

                (#3! ((find (compose cq- ! ant act lex) ~verb))
                 (list #3! ((find (compose cq- ! ant act lex) ~verb))))

        (#3! ((find (compose cq- ! &ant act lex) ~verb))
                 (list #3! ((find (compose cq- ! &ant act lex) ~verb))))

                (#3! ((deduce cause (build agent $vsub11 act (build lex ~verb))
                                        effect $result))
                 (list #3! ((find (compose effect- cause act lex) ~verb))))))




;-------------------------------------------------------------------------
;
;               function: effect
;               input:          a verb to be defined
;               output: a list containing the enabling conditions
;                               of .  List will contain either propositions (molecular
;                               nodes) or patterns for them, from rules (pattern node).
;-------------------------------------------------------------------------
(defun effect (verb)
   (cond  (#3! ((find (compose ant- ! cq act lex) ~verb))
                 (list #3! ((find (compose ant- ! cq act lex) ~verb))))
          (#3! ((find (compose &ant- ! cq act lex) ~verb))
                 (list #3! ((find (compose &ant- ! cq act lex) ~verb))))))

;-------------------------------------------------------------------------

(defun prim_base (verb) '(not set yet))

;-------------------------------------------------------------------------
;
;               function: emptyp  (a predicate)
;               input: a list
;               output: t if the input list is empty, or a list* of empty lists,
;                               nil if list contains any elements which are non-null.
;-------------------------------------------------------------------------
(defun emptyp (lst)
    (cond ((null lst) t)
          ((AND (listp lst) (emptyp (car lst))) (emptyp (cdr lst)))))

(defun cathelper (lst aset)
  (cond ((null lst) aset)
                (t (cathelper (cdr lst) (adjoin (car lst) aset)))))
;-------------------------------------------------------------------------
```

8.2 My Improved Version of Verb Algorithm

```
(in-package :snepsul)

(setq verb_trace NIL)
(setq subject_trace NIL)
(setq object_trace NIL)
(setq ind_object_trace NIL)
(setq cause_trace NIL)
(setq effect_trace NIL)


;-------------------------------------------------------------------------
;
;           function: defn_verb
;           input:    a verb to be defined
;           output: predicate structure of , with categorization of arguments;
;                     causal/enablement information; eventually to include primitive
;                     act of which  is a type (if any).
;           calls:  report_bitransitive, or report_transitive, or report_reflexive,
;                     or report_intransitve, as appropriate.
;           NOTE: #3! is a macro which allows snepsul commands to be invoked from
;                within lisp functions; obviates need for references to pkgs, etc.
;           *******modified RMS 4-02******
;-------------------------------------------------------------------------
(defun defn_verb (verb)
  (setq czs (cause verb))
  (setq efs (car (effect verb)))
  (setq subject (categorize_subject verb))
  (setq object (categorize_object verb))
  (setq ind_obj (categorize_indobject verb))
  (if verb_trace
     (list (report verb czs efs subject object ind_obj)
           (report_trace))
   (report verb czs efs subject object ind_obj)))

;-------------------------------------------------------------------------
;
;           function: report
;           input: a verb to be defined
;           output: predicate structure of , with categorization of arguments;
;                     causal/enablement information; eventually to include primitive
;                     act of which  is a type (if any).
;           calls:    categorize_subject, categorize_object, categorize_indobject,
;                     cause, effect, (prim_base currently undefined)
;   **********added by RMS 4-02**********
;-------------------------------------------------------------------------
(defun report (verb czs efs sub obj ind)
  (setq result (list 'a sub 'can 'verb))
  (if obj
     (if ind
        (setq result (append
                   result
                   (list 'a obj 'to 'a ind)))
      (setq result (append
```

```
                result
                (list 'a obj)))))
  (setq result (append
             result
             (list 'result= #3!((describe ~czs))
                 'enabled 'by= #3!((describe ~efs)))))
  ;           (prim_base verb)
  )

;---------------------------------------------------------------------------
;
;           function: report_trace
;           input: nil
;           output: which functions returned information about the verb
;           calls:nil
;   ************added by RMS 4-02**********
;---------------------------------------------------------------------------
(defun report_trace ()
  (list subject_trace object_trace ind_object_trace cause_trace effect_trace))



;---------------------------------------------------------------------------
;
;           function: categorize_subject
;           input:  a verb to be defined
;           output: categorization of encountered subjects of  as (in order
;                       of preference) 1)belonging to some basic level category,
;                       2)belonging to some subclass of animal, or 3)belonging to some
;                       miscellaneous (but known) class.
;           calls:      base_cat_subj, anim_subj, some_cat_subj, emptyp
           *********modified RMS 4-02**********
;---------------------------------------------------------------------------
(defun categorize_subject (verb)
  (setq base_subj (base_cat_subj verb))
  (setq animal_subj (anim_subj verb))
  (setq some_subj (some_cat_subj verb))

  (cond ((emptyp (list base_subj animal_subj))
        (setq subject some_subj)
        (if verb_trace
           (setq subject_trace (list 'subject 'found 'by 'some_cat_subj)))
        (if (listp subject)
           (cathelper subject nil)
          subject))
       ((emptyp base_subj)
        (if verb_trace
           (setq subject_trace (list 'subject 'found 'by 'anim_subj)))
        (cathelper animal_subj nil))
       (t (if verb_trace
             (setq subject_trace (list 'subject 'found 'by 'base_cat_subj)))
          (cathelper base_subj nil))))

;---------------------------------------------------------------------------
;
;           function: person_subj  (function not currently used in defining verb)
```

```
;           input:  a verb to be defined
;           output: the atom 'person, if a member of the class person has been
;                       encountered as the subject of
;-------------------------------------------------------------------------------
(defun person_subj (verb)
    (cond ((AND #3! ((deduce agent $vsub act (build lex ~verb)))
                    #3! ((deduce member *vsub class (build lex "person"))))
           'person)))
;-------------------------------------------------------------------------------
;
;           function: anim_subj
;           input:  a verb to be defined
;           output: a list of the kinds of animal which have been known to
;-------------------------------------------------------------------------------
(defun anim_subj (verb)
    (cond ((AND #3! ((deduce agent $vsub1 act (build lex ~verb)))
                    #3! ((deduce member *vsub1 class (build lex "animal"))))
          (append #3! ((find (compose lex- class- ! member
                                agent- ! act lex) ~verb))
                  #3! ((find (compose lex- class- ! members
                                agent- ! act lex) ~verb))))
;          (#3! ((find (compose lex- class-  member
;                              agent-  act lex) ~verb))
;           (list #3! ((find (compose lex- class-  member
;                               agent-  act lex) ~verb))))
))



;-------------------------------------------------------------------------------
;
;           function: base_cat_subj
;           input:  a verb to be defined
;           output: a list of basic level categs. which have been known to
;-------------------------------------------------------------------------------
(defun base_cat_subj (verb)
    (cond ((AND #3! ((deduce agent $vsub2 act (build lex ~verb)))
                    (OR #3! ((deduce member *vsub2 class $vbasic))
                        #3! ((deduce members *vsub2 class $vbasic)))
                    #3! ((deduce object1 *vbasic rel "ISA"
                    object2 (build lex "basic ctgy"))))
          (append #3! ((find (compose lex- class- ! member
                                agent- ! act lex) ~verb))
                  #3! ((find (compose lex- class- ! members
                                agent- ! act lex) ~verb))))))
;-------------------------------------------------------------------------------
;
;           function: some_cat_subj
;           input:  a verb to be defined
;           output: a list of the kinds of things which have been known to
;                       or the atom 'something, if nothing known about what can .
;-------------------------------------------------------------------------------
(defun some_cat_subj (verb)
    (cond ((AND #3! ((deduce agent $vsub3 act (build lex ~verb)))
                  (OR #3! ((deduce object1 *vsub3 rel "ISA" object2 $somecat))
                      #3! ((deduce objects1 *vsub3 rel "ARE" object2 $somecat))))
```

```
          (append #3! ((find (compose lex- object2- ! object1
                                              agent- ! act lex) ~verb))
                #3! ((find (compose lex- object2- ! objects1
                                              agent- ! act lex) ~verb))))
;         (#3! ((find (compose lex- object2-  object1
;                                  agent-  act lex) ~verb))
;          (list #3! ((find (compose lex- object2-  object1
;                                              agent-  act lex) ~verb))))
          (t 'something)))



;---------------------------------------------------------------------------
;
;           function: categorize_object
;           input:  a verb to be defined
;           output: categorization of encountered objects of  as (in order
;                     of preference) 1)belonging to some basic level category,
;                     2)belonging to some subclass of animal, or 3)belonging to some
;                     miscellaneous (but known) class.
;           calls:     base_cat_obj, anim_obj, some_cat_obj, emptyp
;    **********modified RMS 4-02***************
;---------------------------------------------------------------------------
(defun categorize_object (verb)
  (setq base_obj (base_cat_obj verb))
  (setq animal_obj (anim_obj verb))
  (setq some_obj (some_cat_obj verb))

  (cond ((emptyp (list base_obj animal_obj))
         (setq object some_obj)
         (if verb_trace
            (setq object_trace (list 'object 'found 'by 'some_cat_obj)))
         (if (listp object)
            (cathelper object nil)
           object))
        ((emptyp base_obj)
         (if verb_trace
            (setq object_trace (list 'object 'found 'by 'anim_obj)))
         (cathelper animal_obj nil))
        (t (if verb_trace
              (setq object_trace (list 'object 'found 'by 'base_cat_obj)))
           (cathelper base_obj nil))))

;---------------------------------------------------------------------------
;
;           function: person_obj  (function not currently used in defining verb)
;           input:  a verb to be defined
;           output: the atom 'person, if a member of the class person has been
;                     encountered as the object of
;---------------------------------------------------------------------------
(defun person_obj (verb)
    (cond ((AND #3! ((deduce object $vobj agent $vsub4 act (build lex ~verb)))
                     #3! ((deduce member *vobj class (build lex "person"))))
              'person)))
```

```
;-------------------------------------------------------------------------------
;
;           function: base_cat_obj
;           input:  a verb to be defined
;           output: a list of basic categs. of things known to have been ed.
;-------------------------------------------------------------------------------
(defun base_cat_obj (verb)
    (cond ((AND #3! ((deduce object $vobj1 agent $vsub5 act (build lex ~verb)))
                    #3! ((deduce member *vobj1 class $vbasic1))
                    #3! ((deduce object1 *vbasic1 rel "ISA"
                                object2 (build lex "basic ctgy"))))
          (append #3! ((find (compose lex- class- ! member
                                        object- ! act lex) ~verb))
                #3! ((find (compose lex- class- ! members
                                        object- ! act lex) ~verb))))
;         (#3! ((find (compose lex- class-  member
;                                       object-  act lex) ~verb))
;          (list #3! ((find (compose lex- class-  member
;                                       object-  act lex) ~verb))))
))
;-------------------------------------------------------------------------------
;
;           function: anim_obj
;           input:  a verb to be defined
;           output: a list of the kinds of animals known to have been ed.
;-------------------------------------------------------------------------------
(defun anim_obj (verb)
    (cond ((AND #3! ((deduce agent $vsub6 object $vobj2 act (build lex ~verb)))
                    #3! ((deduce member *vobj2 class (build lex "animal"))))
          (append #3! ((find (compose lex- class- ! member
                                        object- ! act lex) ~verb))
                #3! ((find (compose lex- class- ! members
                                        object- ! act lex) ~verb))))))



;-------------------------------------------------------------------------------
;
;           function: some_cat_obj
;           input:  a verb to be defined
;           output: a list of the kinds of things known to have been ed,
;                       or the atom 'something, if nothing known about what can
;                       be ed.
; *********Modified RMS 4-02****************
;-------------------------------------------------------------------------------

(defun some_cat_obj (verb)
  (cond (#3! ((deduce agent $vsub7 object $vobj3 act (build lex ~verb)))
          (cond ((OR #3! ((deduce object1 *vobj3 rel "ISA" object2 *somecat))
                     #3! ((deduce objects1 *vobj3 rel "ARE" object2 *somecat)))
                (append #3! ((find (compose lex- object2- ! object1
                                        object- ! act lex) ~verb))
                      #3! ((find (compose lex- object2- ! objects1
                                        object- ! act lex) ~verb))))
;         (#3! ((find (compose lex- object2-  object1
```

```
;                                          agent-  act lex) ~verb))
;          (list #3! ((find (compose lex- object2-  object1
;                                          agent-  act lex) ~verb))))
               (t 'something)))
      (t nil)
))



;-------------------------------------------------------------------------
;
;           function: categorize_indobject
;           input:  a verb to be defined
;           output: categorization of encountered indirect objects of  as
;                     (in order of preference) 1)belonging to some basic level
;                     category, 2)belonging to some subclass of animal, or
;                     3)belonging to some miscellaneous (but known) class.
;           calls:     base_cat_indobj, anim_indobj, some_cat_indobj, emptyp
;  ********************modified RMS 4-02**********************
;-------------------------------------------------------------------------
(defun categorize_indobject (verb)
  (setq base_ind_obj (base_cat_indobj verb))
  (setq animal_ind_obj (anim_indobj verb))
  (setq some_ind_obj (some_cat_indobj verb))

  (cond ((emptyp (list base_ind_obj animal_ind_obj))
        (setq ind_object some_ind_obj)
        (if verb_trace
           (setq ind_object_trace (list 'indirect 'object 'found 'by 'some_cat_indobj)))
        (if (listp ind_object)
           (cathelper ind_object nil)
          ind_object))
       ((emptyp base_ind_obj)
        (if verb_trace
           (setq ind_object_trace (list 'indirect 'object 'found 'by 'anim_indobj)))
        (cathelper animal_ind_obj nil))
       (t (if verb_trace
             (setq ind_object_trace (list 'indirect 'object 'found 'by 'base_cat_indobj)))
          (cathelper base_ind_obj nil))))


;-------------------------------------------------------------------------
;
;           function: person_indobj  (function not currently used in defining verb )
;           input:  a verb to be defined
;           output: the atom 'person, if a member of the class person has been
;                     encountered as the indirect object of
;-------------------------------------------------------------------------
(defun person_indobj (verb)
    (cond ((AND #3! ((deduce indobj $vindobj object $vobj4
                                    agent $vsub8 act (build lex ~verb)))
                    #3! ((deduce member *vindobj class (build lex "person"))))
              'person)))
```

```
;-------------------------------------------------------------------------
;
;           function: base_cat_indobj
;           input:  a verb to be defined
;           output: a list of basic categs. of things encountered as indirect
;                   object of
;-------------------------------------------------------------------------
(defun base_cat_indobj (verb)
    (cond ((AND #3! ((deduce object $vobj5 agent $vsub9
                                    indobj $vindobj1 act (build lex ~verb)))
                    #3! ((deduce member *vindobj1 class $vbasic1))
                    #3! ((deduce object1 *vbasic1 rel "ISA"
                    object2 (build lex "basic ctgy"))))
          (append #3! ((find (compose lex- class- ! member
                                    indobj- ! act lex) ~verb))
                #3! ((find (compose lex- class- ! members
                                    indobj- ! act lex) ~verb))))
;          (#3! ((find (compose lex- class- member
;                                    indobj- act lex) ~verb))
;           (list #3! ((find (compose lex- class- member
;                                    indobj- act lex) ~verb))))
))
;-------------------------------------------------------------------------
;
;           function: anim_indobj
;           input:  a verb to be defined
;           output: a list of the kinds of animals known to have been the indirect
;                   object of .
;-------------------------------------------------------------------------
(defun anim_indobj (verb)
    (cond ((AND #3! ((deduce agent $vsub9 object $vobj6
                                    indobj $vindobj2 act (build lex ~verb)))
                    #3! ((deduce member *vindobj2 class (build lex "animal"))))
          (append #3! ((find (compose lex- class- ! member
                                    indobj- ! act lex) ~verb))
                #3! ((find (compose lex- class- ! members
                                    indobj- ! act lex) ~verb))))
))



;-------------------------------------------------------------------------
;
;           function: some_cat_indobj
;           input:  a verb to be defined
;           output: a list of the kinds of things known to have been the indirect
;                   object of , if any;
;                   the atom 'something if nothing known about what can be the
;                   indirect object of
; *********Modified RMS 4-02****************
;-------------------------------------------------------------------------
(defun some_cat_indobj (verb)
  (cond (#3! ((deduce agent $vsub10 object $vobj7
                  indobj $vindobj3 act (build lex ~verb)))
          (cond ((OR #3! ((deduce object1 *vindobj3 rel "ISA" object2 *somecat))
```

```
                        #3! ((deduce objects1 *vindobj3 rel "ARE" object2 *somecat)))
            (append #3! ((find (compose lex- object2- ! object1
                                indobj- ! act lex) ~verb))
                    #3! ((find (compose lex- object2- ! objects1
                                indobj- ! act lex) ~verb))))
;         (#3! ((find (compose lex- object2- object1
;                                indobj- act lex) ~verb))
;         (list #3! ((find (compose lex- object2- object1
;                                indobj- act lex) ~verb))))
              (t 'something)))
      (t nil)
))



;-------------------------------------------------------------------------
;
;           function: cause
;           input:     a verb to be defined
;           output: a list containing the result of .  List will contain
;           either propositions (molecular nodes) or patterns for them,
;           from rules (pattern node).
;   **************modified RMS 4-02*****************
;-------------------------------------------------------------------------
(defun cause (verb)
  (setq cz1 #3! ((find (compose cq- ! ant act lex) ~verb)))
  (setq cz2 #3! ((find (compose cq- ! &ant act lex) ~verb)))

  (cond (#3! ((deduce object1 (build agent *vsub11 act (build lex ~verb))
                          rel (build lex "enable")
                          object2 $goal))
              (cond ((and cz1 cz2)
                          (if verb_trace
                                  (setq cause_trace '(cause found both enabled verbs
                                                  and cqs from ants and &ants)))
                          (list cz1 'or 'cz2 'or 'to 'enable
                                  #3! ((find (compose lex- act- object2- ! rel lex) "enable"
                                                  (compose lex- act- object2- ! object1 act lex)
                                                  ~verb))))
                         (cz1
                          (if verb_trace
                                  (setq cause_trace '(cause found both enabled verbs
                                                  and cqs from ants)))
                          (list cz1 'or 'to 'enable
                                  #3! ((find (compose lex- act- object2- ! rel lex) "enable"
                                                  (compose lex- act- object2- ! object1 act lex)
                                                  ~verb))))
                         (cz2
                          (if verb_trace
                                  (setq cause_trace '(cause found both enabled verbs
                                                  and cqs from &ants)))
                          (list cz2 'or 'to 'enable
                                  #3! ((find (compose lex- act- object2- ! rel lex) "enable"
                                                  (compose lex- act- object2- ! object1 act lex)
                                                  ~verb))))
                         (t
```

```
            (if verb_trace
                  (setq cause_trace '(cause found enabled verbs)))
            (list 'to 'enable
                  #3! ((find (compose lex- act- object2- ! rel lex)
                             "enable"
                             (compose lex- act- object2- ! object1 act lex)
                             ~verb))))))

(#3! ((deduce mode (build lex "presumably")
              object (build object1 (build agent *vsub11
                                           act (build lex ~verb)
                                           time $vtime)
                            rel (build lex "enable")
                            object2 $goal)))
   (cond ((and cz1 cz2)
            (if verb_trace
                  (setq cause_trace '(cause found both presumably
                                            enabled verbs and cqs from
                                            ants and &ants)))
            (list cz1 'or cz2
                  'or 'to 'enable
                  #3! ((find (compose lex- act- object2- rel lex) "enable"
                             (compose lex- act- object2- object1 act lex) ~verb
                             (compose lex- act- object2- object- ! mode lex)
                             "presumably"))))
           (cz1
            (if verb_trace
                  (setq cause_trace '(cause found both presumably
                                            enabled verbs and cqs from
                                            ants)))
            (list cz1
                  'or 'to 'enable
                  #3! ((find (compose lex- act- object2- rel lex) "enable"
                             (compose lex- act- object2- object1 act lex) ~verb
                             (compose lex- act- object2- object- ! mode lex)
                             "presumably"))))
           (cz2
            (if verb_trace
                  (setq cause_trace '(cause found both presumably
                                            enabled verbs and cqs from
                                            &ants)))
            (list cz2
                  'or 'to 'enable
                  #3! ((find (compose lex- act- object2- rel lex) "enable"
                             (compose lex- act- object2- object1 act lex) ~verb
                             (compose lex- act- object2- object- ! mode lex)
                             "presumably"))))
           (t
            (if verb_trace
                  (setq cause_trace '(cause found presumably enabled verbs)))
            (list 'to 'enable
                  #3! ((find (compose lex- act- object2- rel lex) "enable"
                             (compose lex- act- object2- object1 act lex) ~verb
                             (compose lex- act- object2- object- ! mode lex)
                             "presumably"))))))
```

```
            ((and cz1 cz2)
             (if verb_trace
                (setq cause_trace '(cause found cqs from ants and &ants)))
             (list cz1 cz2))

            (cz1
             (if verb_trace
                (setq cause_trace '(cause found cqs from ants)))
             (list cz1))

            (cz2
             (if verb_trace
                (setq cause_trace '(cause found cqs from &ants)))
             (list cz2))

            (#3! ((deduce cause (build agent $vsub11 act (build lex ~verb))
                          effect $result))
                (if verb_trace
                        (setq cause_trace '(cause found effects)))
                (list #3! ((find (compose effect- cause act lex) ~verb))))
            (t
             (if verb_trace
                (setq cause_trace '(no causes found)))
             NIL)))

;-------------------------------------------------------------------------
;
;           function: effect
;           input:    a verb to be defined
;           output: a list containing the enabling conditions
;                     of .  List will contain either propositions (molecular
;                     nodes) or patterns for them, from rules (pattern node).
;*******************modified RMS 4-02****************
;-------------------------------------------------------------------------
(defun effect (verb)
  (setq ef1 #3! ((find (compose ant- ! cq act lex) ~verb)))
  (setq ef2 #3! ((find (compose &ant- ! cq act lex) ~verb)))

  (cond  ((and ef1 ef2)
             (if verb_trace
                (setq effect_trace '(effects found at ants and &ants)))
             (list ef1 ef2))
            (ef1
             (if verb_trace
                (setq effect_trace '(effects found at ants)))
             (list ef1))
            (ef2
             (if verb_trace
                (setq effect_trace '(effects found at &ants)))
             (list ef2))
            (t
             (if verb_trace
                (setq effect_trace '(no effects found)))
             NIL)))

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

```
(defun prim_base (verb) '(not set yet))

;-----------------------------------------------------------------------------
;
;           function: emptyp  (a predicate)
;           input: a list
;           output: t if the input list is empty, or a list* of empty lists,
;                       nil if list contains any elements which are non-null.
;----------------------------------------------------------------------------
(defun emptyp (lst)
    (cond ((null lst) t)
          ((AND (listp lst) (emptyp (car lst))) (emptyp (cdr lst)))))

(defun cathelper (lst aset)
   (cond ((null lst) aset)
             (t (cathelper (cdr lst) (adjoin (car lst) aset)))))
;-----------------------------------------------------------------------------
```

8.3 Justin Del Vecchio's throw demo

This demo was written by Justin Del Vecchio and is used here with his permission.

```
;;; Reset the network
(resetnet t)

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

;;; Introduce Marc's relations
(intext "rels")

;;; Compose paths
(intext "paths")

;;; Load Noun/Verb Algorithm
;;;one of the following two lines is chosen depending on whether your
;;;choice of algorithm is the original (verb) or my version (fast_verb_fixed_well)
;;;^(load "fast_verb_fixed_well")
^(load "verb")

;;; turn on defn_verb trace
^(setq verb_trace t)

;; Define the relations necessary
(define lex act agent member class object1 rel object2 property object indobj cause effect kn_cat)


;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;; Example One - Demonstration of Ehrlich's verb algorithm with a verb not labeled as
;; bitransitive, transitive, etc.
```

```
;; The sentence:
;; "Derek threw the baseball."
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

;; Include background knowledge about a baseball.  Assume that ball is its parent class
;; and that ball is basic level.
(describe (assert subclass (build lex "baseball")
                  superclass (build lex "ball") ))

;; Make reference to the baseball without specifying which baseball it is.
(describe (assert member #baseball
                  class (build lex "baseball")))

;; "Derek threw the baseball."
;; Represent the sentence without making reference to the fact that it is, at first
;; appearance, a transitive verb.  This shows how Ehrlich's algorithim treats verbs
;; not labeled as bitransitive, transitive, etc.

(describe (assert act (build lex "throw")
                  agent (build lex "Derek")
                  object *baseball))


;; Model the fact that ball is a basic level category.  This follows Ehrlich's case frames.
(describe (assert object1 (build lex "baseball")
                          rel ("ISA") object2
                          (build lex "basic ctgy")))

;; Include background knowledge about Derek.  Assume that his parent class is human and
;; that human is a basic level.
(describe (assert member (build lex "Derek")
                          class (build lex "human")))

;; Model the fact that human is a basic level category.  This follows Ehrlich's case
;; frames
(describe (assert object1 (build lex "human")
                  rel ("ISA")
                  object2 (build lex "basic ctgy")))

;; The moment of truth...
^(defn_verb 'throw)

(resetnet)

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;; Example Two - Demonstration of Ehrlich's verb algorithm with a verb that is labeled
;; as transitive.
;; The sentence:
;; "Derek threw the baseball."
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

;; Include background knowledge about a baseball.  Assume that ball is its parent class
;; and that ball is basic level.
(describe (assert subclass (build lex "baseball")
                  superclass (build lex "ball") ))
```

```
;; Make reference to the baseball without specifying which baseball it is.
(describe (assert member #baseball
                  class (build lex "baseball")))

;; "Derek threw the baseball."
;; Represent the sentence without making reference to the fact that it is, at first
;; appearance, a transitive verb.  This shows how Ehrlich's algorithim treats verbs
;; not labeled as bitransitive, transitive, etc.

(describe (assert act (build lex "throw")
                  agent (build lex "Derek")
                  object *baseball))


;; Model the fact that ball is a basic level category.  This follows Ehrlich's case frames.
(describe (assert object1 (build lex "baseball")
                  rel ("ISA")
                  object2 (build lex "basic ctgy")))

;; Include background knowledge about Derek.  Assume that his parent class is human and
;; that human is a basic level.
(describe (assert member (build lex "Derek")
                  class (build lex "human")))

;; Model the fact that human is a basic level category.  This follows Ehrlich's case
;; frames
(describe (assert object1 (build lex "human")
                  rel ("ISA")
                  object2 (build lex "basic ctgy")))

;; Define throw as a transitive verb (at least in the context of the sentence)
(describe (assert object (build lex "throw")
                  property (build lex "transitive")))

;; The moment of truth...
^(defn_verb 'throw)

(resetnet)


;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;; Example Three - Demonstration of Ehrlich's verb algorithm with a verb that is
;; labeled as transitive when it is really bitransitive.
;; The sentence:
;; "Derek threw the baseball to Tino."
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

;; Include background knowledge about a baseball.  Assume that ball is its parent class
;; and that ball is basic level.
(describe (assert subclass (build lex "baseball")
                  superclass (build lex "ball") ))

;; Make reference to the baseball without specifying which baseball it is.
(describe (assert member #baseball
                  class (build lex "baseball")))
```

```
;; "Derek threw the baseball."
;; Represent the sentence without making reference to the fact that it is, at first
;; appearance, a transitive verb.  This shows how Ehrlich's algorithim treats verbs
;; not labeled as bitransitive, transitive, etc.

(describe (assert act (build lex "throw")
                  agent (build lex "Derek")
                  object *baseball
                  indobj(build lex "Tino")))

;; Model the fact that ball is a basic level category.  This follows Ehrlich's case frames.
(describe (assert object1 (build lex "baseball")
                  rel ("ISA")
                  object2 (build lex "basic ctgy")))

;; Include background knowledge about Derek.  Assume that his parent class is human and
;; that human is a basic level.
(describe (assert member (build lex "Derek")
                  class (build lex "human")))

;; Include background knowledge about Tino.  Assume that his parent class is human and
;; that human is a basic level.
(describe (assert member (build lex "Tino")
                  class (build lex "human")))

;; Model the fact that human is a basic level category.  This follows Ehrlich's case
;; frames
(describe (assert object1 (build lex "human")
                  rel ("ISA")
                  object2 (build lex "basic ctgy")))

;; Define throw as a transitive verb (at least in the context of the sentence)
(describe (assert object (build lex "throw")
                  property (build lex "transitive")))

;; The moment of truth...
^(defn_verb 'throw)

(resetnet)

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;; Example Four - Demonstration of Ehrlich's verb algorithm with a verb that is
;; correctly labeled as bitransitive
;; The sentence:
;; "Derek threw the baseball to Tino."
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

;; Include background knowledge about a baseball.  Assume that ball is its parent class
;; and that ball is basic level.
(describe (assert subclass (build lex "baseball")
                  superclass (build lex "ball") ))

;; Make reference to the baseball without specifying which baseball it is.
(describe (assert member #baseball
                  class (build lex "baseball")))
```

```
;; "Derek threw the baseball."
;; Represent the sentence without making reference to the fact that it is, at first
;; appearance, a transitive verb.  This shows how Ehrlich's algorithim treats verbs
;; not labeled as bitransitive, transitive, etc.

(describe (assert act (build lex "throw")
                  agent (build lex "Derek")
                  object *baseball
                  indobj(build lex "Tino")))

;; Model the fact that ball is a basic level category.  This follows Ehrlich's case frames.
(describe (assert object1 (build lex "baseball")
                  rel ("ISA")
                  object2 (build lex "basic ctgy")))

;; Include background knowledge about Derek.  Assume that his parent class is human and
;; that human is a basic level.
(describe (assert member (build lex "Derek")
                  class (build lex "human")))

;; Include background knowledge about Tino.  Assume that his parent class is human and
;; that human is a basic level.
(describe (assert member (build lex "Tino")
                  class (build lex "human")))

;; Model the fact that human is a basic level category.  This follows Ehrlich's case
;; frames
(describe (assert object1 (build lex "human")
                  rel ("ISA")
                  object2 (build lex "basic ctgy")))

;; Define throw as a transitive verb (at least in the context of the sentence)
(describe (assert object (build lex "throw")
                  property (build lex "bitransitive")))

;; The moment of truth...
^(defn_verb 'throw)

(resetnet)

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;; Example Five - Demonstration of Ehrlich's verb algorithm with multiple instances
;; of the same verb.
;; The sentences are:
;; "Derek threw the baseball to Tino."
;; "Bobo throws the rubber ball."
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

;; Include background knowledge about a baseball.  Assume that ball is its parent class
;; and that ball is basic level.
(describe (assert subclass (build lex "baseball")
                  superclass (build lex "ball") ))

;; Make reference to the baseball without specifying which baseball it is.
(describe (assert member #baseball
                  class (build lex "baseball")))
```

```
;; "Derek threw the baseball."
;; Represent the sentence without making reference to the fact that it is, at first
;; appearance, a transitive verb.  This shows how Ehrlich's algorithim treats verbs
;; not labeled as bitransitive, transitive, etc.

(describe (assert act (build lex "throw")
                      agent (build lex "Derek")
                      object *baseball
                      indobj(build lex "Tino")))

;; Model the fact that ball is a basic level category.  This follows Ehrlich's case frames.
(describe (assert object1 (build lex "baseball")
                      rel ("ISA")
                      object2 (build lex "basic ctgy")))

;; Include background knowledge about Derek.  Assume that his parent class is human and
;; that human is a basic level.
(describe (assert member (build lex "Derek")
                      class (build lex "human")))

;; Include background knowledge about Tino.  Assume that his parent class is human and
;; that human is a basic level.
(describe (assert member (build lex "Tino")
                      class (build lex "human")))

;; Model the fact that human is a basic level category.  This follows Ehrlich's case
;; frames
(describe (assert object1 (build lex "human")
                      rel ("ISA")
                      object2 (build lex "basic ctgy")))

;; Define throw as a transitive verb (at least in the context of the sentence)
(describe (assert object (build lex "throw")
                      property (build lex "bitransitive")))

;; "Bobo throws the rubber ball"
;; Here assume that Bobo is a chimp.  Assume that chimp is a basic level category as well
;; as rubber ball.
;; Note that the sentence uses throw in a transitive sense.  Will the algoritm make a distinction
;; between the type of throwing a human can do and that which a chimp can do?

;; Include background knowledge about a rubber ball.  Assume that ball is its parent class.
(describe (assert subclass (build lex "rubber ball")
                      superclass (build lex "ball") ))

;; Make reference to the baseball without specifying which baseball it is.
(describe (assert member #rubberBall
                      class (build lex "rubber ball")))

;; Model the action.
(describe (assert act (build lex "throw")
                      agent (build lex "Bobo")
                      object *rubberBall
                      ))
```

```
;; Model the fact that rubberball is a basic level category.
;; This follows Ehrlich's case frames.
(describe (assert object1 (build lex "rubber ball")
                  rel ("ISA")
                  object2 (build lex "basic ctgy")))

;; Model that Bobo is a chimp.
(describe (assert member (build lex "Bobo")
                  class (build lex "chimp")))

;; Model that chimp is a basic level category.
(describe (assert object1 (build lex "chimp")
                  rel ("ISA")
                  object2 (build lex "basic ctgy")))

;; The moment of truth...
^(defn_verb 'throw)

(resetnet)

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;; Example Six - Demonstration of Ehrlich's verb algorithm with a verb used multiple
;; times in both transitive and bitransitive forms.  However, the transitive and
;; bitransitive lexes are set for the verb each time!  Will Ehrlich's algorithm
;; differentiate between the two?
;; The sentences are:
;; "Derek threw the baseball to Tino."
;; "Bobo throws the rubber ball."
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

;; Include background knowledge about a baseball.  Assume that ball is its parent class
;; and that ball is basic level.
(describe (assert subclass (build lex "baseball")
                  superclass (build lex "ball") ))

;; Make reference to the baseball without specifying which baseball it is.
(describe (assert member #baseball
                  class (build lex "baseball")))

;; "Derek threw the baseball."
;; Represent the sentence without making reference to the fact that it is, at first
;; appearance, a transitive verb.  This shows how Ehrlich's algorithim treats verbs
;; not labeled as bitransitive, transitive, etc.

(describe (assert act (build lex "throw")
                  agent (build lex "Derek")
                  object *baseball
                  indobj(build lex "Tino")))

;; Model the fact that ball is a basic level category.  This follows Ehrlich's case frames.
(describe (assert object1 (build lex "baseball")
                  rel ("ISA")
                  object2 (build lex "basic ctgy")))

;; Include background knowledge about Derek.  Assume that his parent class is human and
;; that human is a basic level.
```

```
(describe (assert member (build lex "Derek")
                  class (build lex "human")))

;; Include background knowledge about Tino.  Assume that his parent class is human and
;; that human is a basic level.
(describe (assert member (build lex "Tino")
                  class (build lex "human")))

;; Model the fact that human is a basic level category.  This follows Ehrlich's case
;; frames
(describe (assert object1 (build lex "human")
                  rel ("ISA")
                  object2 (build lex "basic ctgy")))

;; Define throw as a transitive verb (at least in the context of the sentence)
(describe (assert object (build lex "throw")
                  property (build lex "bitransitive")))

;; "Bobo throws the rubber ball"
;; Here assume that Bobo is a chimp.  Assume that chimp is a basic level category as well
;; as rubber ball.
;; Note that the sentence uses throw in a transitive sense.  Will the algoritm make a distinction
;; between the type of throwing a human can do and that which a chimp can do?

;; Include background knowledge about a rubber ball.  Assume that ball is its parent class.
(describe (assert subclass (build lex "rubber ball")
                  superclass (build lex "ball") ))

;; Make reference to the baseball without specifying which baseball it is.
(describe (assert member #rubberBall
                  class (build lex "rubber ball")))

;; Model the action.
(describe (assert act (build lex "throw")
                  agent (build lex "Bobo")
                  object *rubberBall
                  ))

;; Define throw as a transitive verb.  Now it has been defined in two ways.  Will this affect
;; Ehrlich's algorithm?
(describe (assert object (build lex "throw")
                  property (build lex "transitive")))

;; Model the fact that rubberball is a basic level category.
;; This follows Ehrlich's case frames.
(describe (assert object1 (build lex "rubber ball")
                  rel ("ISA")
                  object2 (build lex "basic ctgy")))

;; Model that Bobo is a chimp.
(describe (assert member (build lex "Bobo")
                  class (build lex "chimp")))

;; Model that chimp is a basic level category.
(describe (assert object1 (build lex "chimp")
                  rel ("ISA")
```

```
                    object2 (build lex "basic ctgy")))

;; The moment of truth...
^(defn_verb 'throw)

(resetnet)

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;; Example Seven - Demonstrate Ehrlich's algorithm with a verb that has a consequence
;; defined for it.
;; The sentences are:
;; "Derek threw the baseball to Tino."
;; "Bobo throws the rubber ball."
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

;; Include background knowledge about a baseball.  Assume that ball is its parent class
;; and that ball is basic level.
(describe (assert subclass (build lex "baseball")
                    superclass (build lex "ball") ))

;; Make reference to the baseball without specifying which baseball it is.
(describe (assert member #baseball
                    class (build lex "baseball")))

;; "Derek threw the baseball."
;; Represent the sentence without making reference to the fact that it is, at first
;; appearance, a transitive verb.  This shows how Ehrlich's algorithim treats verbs
;; not labeled as bitransitive, transitive, etc.

(describe (assert act (build lex "throw")
                    agent (build lex "Derek")
                    object *baseball
                    indobj(build lex "Tino")))

;; Model the fact that ball is a basic level category.  This follows Ehrlich's case frames.
(describe (assert object1 (build lex "baseball")
                    rel ("ISA")
                    object2 (build lex "basic ctgy")))

;; Include background knowledge about Derek.  Assume that his parent class is human and
;; that human is a basic level.
(describe (assert member (build lex "Derek")
                    class (build lex "human")))

;; Include background knowledge about Tino.  Assume that his parent class is human and
;; that human is a basic level.
(describe (assert member (build lex "Tino")
                    class (build lex "human")))

;; Model the fact that human is a basic level category.  This follows Ehrlich's case
;; frames
(describe (assert object1 (build lex "human")
                    rel ("ISA")
                    object2 (build lex "basic ctgy")))

;; Define throw as a transitive verb (at least in the context of the sentence)
```

```
(describe (assert object (build lex "throw")
                property (build lex "bitransitive")))

;; "Bobo throws the rubber ball"
;; Here assume that Bobo is a chimp.  Assume that chimp is a basic level category as well
;; as rubber ball.
;; Note that the sentence uses throw in a transitive sense.  Will the algoritm make a distinction
;; between the type of throwing a human can do and that which a chimp can do?

;; Include background knowledge about a rubber ball.  Assume that ball is its parent class.
(describe (assert subclass (build lex "rubber ball")
                superclass (build lex "ball") ))

;; Make reference to the baseball without specifying which baseball it is.
(describe (assert member #rubberBall
                class (build lex "rubber ball")))

;; Model the action.
(describe (assert act (build lex "throw")
                agent (build lex "Bobo")
                object *rubberBall
                ))

;; Define throw as a transitive verb.  Now it has been defined in two ways.  Will this affect
;; Ehrlich's algorithm?
(describe (assert object (build lex "throw")
                property (build lex "transitive")))

;; Model the fact that rubberball is a basic level category.
;; This follows Ehrlich's case frames.
(describe (assert object1 (build lex "rubber ball")
                rel ("ISA")
                object2 (build lex "basic ctgy")))

;; Model that Bobo is a chimp.
(describe (assert member (build lex "Bobo")
                class (build lex "chimp")))

;; Model that chimp is a basic level category.
(describe (assert object1 (build lex "chimp")
                rel ("ISA")
                object2 (build lex "basic ctgy")))

;; Create a path-based inference we will need for subclasses
;; Notice the results this has on the verb algorithms output!
(define-path class
        (compose class
                (kstar (compose subclass- ! superclass))))

;; Create a consequence (in Ehrlichs terms) for throwing.  This consequence is what
;; Ehrlich would have considered a life-rule2
;; "If a person x throws a ball then he lofts it in the air

(describe (assert forall ($h $b)
                &ant((build member *h
                        class (build lex "human"))
```

```
                (build member *b
                        class (build lex "ball"))
                (build agent *h
                        act (build lex "throw")
                        object *b))
            cq (build agent *h
                        act (build lex "loft")
                        object *b
                        location (build lex "through the air")
                        )
           kn_cat ("life-rule.2")))

;; Proof that the rule is in working order
(describe (deduce agent $whoever act (build lex "loft")))


^(defn_verb 'throw)

(resetnet)
```

8.3.1 Results with original algorithm

```
=============================================================
Starting image `/util/acl5.0.1/composer'
  with no arguments
  in directory `/home/csgrad/sood/cse663/demos/'
  on machine `localhost'.

Allegro CL Enterprise Edition 5.0.1 [SPARC] (8/20/99 10:07)
Copyright (C) 1985-1999, Franz Inc., Berkeley, CA, USA.  All Rights Reserved.
;; 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).
USER(1): :ld /projects/snwiz/bin/sneps
; Fast loading /projects/snwiz/bin/sneps.fasl
Loading system SNePS...10% 20% 30% 40% 50% 60% 70% 80% 90% 100%
SNePS-2.5 [PL:1 1999/08/19 16:38:25] loaded.
Type `(sneps)' or `(snepslog)' to get started.
USER(2): (sneps)


   Welcome to SNePS-2.5 [PL:1 1999/08/19 16:38:25]

Copyright (C) 1984--1999 by Research Foundation of
State University of New York. SNePS comes with ABSOLUTELY NO WARRANTY!
Type `(copyright)' for detailed copyright information.
Type `(demo)' for a list of example applications.

   5/6/2002 2:43:27
```

```
* (demo "throw.demo")

File /home/csgrad/sood/cse663/demos/throw.demo is now the source of input.


 CPU time : 0.02

* ;;; Reset the network
(resetnet t)

Net reset


 CPU time : 0.01

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



 CPU time : 0.00

*
;;; Introduce Marc's relations
(intext "rels")
File rels is now the source of input.


 CPU time : 0.00

* _ACT is already defined.
_ACTION is already defined.
_EFFECT is already defined.
_OBJECT1 is already defined.
_OBJECT2 is already defined.

(A1 A2 A3 A4 ACT ACTION AFTER AGENT ANTONYM ASSOCIATED BEFORE CAUSE CLASS
 DIRECTION EFFECT EQUIV ETIME FROM IN INDOBJ INSTR INTO LEX LOCATION KN_CAT
 MANNER MEMBER MEMBERS MODE OBJECT OBJECTS OBJECT1 OBJECTS1 OBJECT2 ON ONTO
 PART PLACE POSSESSOR PROPER-NAME PROPERTY PURPOSE REL SKF STIME SUBCLASS
 SUPERCLASS SYNONYM TIME TO WHOLE)

 CPU time : 0.07

*

End of file rels


 CPU time : 0.07

*
;;; Compose paths
```

```
(intext "paths")
File paths is now the source of input.


 CPU time : 0.00

*
BEFORE implied by the path (COMPOSE BEFORE (KSTAR (COMPOSE AFTER- ! BEFORE)))
BEFORE- implied by the path (COMPOSE (KSTAR (COMPOSE BEFORE- ! AFTER)) BEFORE-)


 CPU time : 0.01

*
AFTER implied by the path (COMPOSE AFTER (KSTAR (COMPOSE BEFORE- ! AFTER)))
AFTER- implied by the path (COMPOSE (KSTAR (COMPOSE AFTER- ! BEFORE)) AFTER-)


 CPU time : 0.00

*
SUB1 implied by the path (COMPOSE OBJECT1- SUPERCLASS- ! SUBCLASS SUPERCLASS-
                         ! SUBCLASS)
SUB1- implied by the path (COMPOSE SUBCLASS- ! SUPERCLASS SUBCLASS- !
                          SUPERCLASS OBJECT1)


 CPU time : 0.00

*
SUPER1 implied by the path (COMPOSE SUPERCLASS SUBCLASS- ! SUPERCLASS OBJECT1-
                           ! OBJECT2)
SUPER1- implied by the path (COMPOSE OBJECT2- ! OBJECT1 SUPERCLASS- ! SUBCLASS
                            SUPERCLASS-)


 CPU time : 0.00

*
SUPERCLASS implied by the path (OR SUPERCLASS SUPER1)
SUPERCLASS- implied by the path (OR SUPERCLASS- SUPER1-)


 CPU time : 0.00

*

End of file paths


 CPU time : 0.01

*
;;; Load Noun/Verb Algorithm
^(
--> load "verb")
```

```
; Fast loading /home/csgrad/sood/cse663/demos/verb.fasl
T



 CPU time : 0.02

*
;; Define the relations necessary
(define lex act agent member class object1 rel object2 property object indobj cause effect kn_cat)
_LEX is already defined.
_ACT is already defined.
_AGENT is already defined.
_MEMBER is already defined.
_CLASS is already defined.
_OBJECT1 is already defined.
_REL is already defined.
_OBJECT2 is already defined.
_PROPERTY is already defined.
_OBJECT is already defined.
_INDOBJ is already defined.
_CAUSE is already defined.
_EFFECT is already defined.
_KN_CAT is already defined.

(LEX ACT AGENT MEMBER CLASS OBJECT1 REL OBJECT2 PROPERTY OBJECT INDOBJ CAUSE
 EFFECT KN_CAT)

 CPU time : 0.01

*

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;; Example One - Demonstration of Ehrlich's verb algorithm with a verb not labeled as
;; bitransitive, transitive, etc.
;; The sentence:
;; "Derek threw the baseball."
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

;; Include background knowledge about a baseball.  Assume that ball is its parent class
;; and that ball is basic level.
(describe (assert subclass (build lex "baseball")
                  superclass (build lex "ball") ))

(M3! (SUBCLASS (M1 (LEX baseball))) (SUPERCLASS (M2 (LEX ball))))

(M3!)

 CPU time : 0.00

*
;; Make reference to the baseball without specifying which baseball it is.
(describe (assert member #baseball
                  class (build lex "baseball")))

(M4! (CLASS (M1 (LEX baseball))) (MEMBER B1))
```

```
(M4!)

 CPU time : 0.00

*
;; "Derek threw the baseball."
;; Represent the sentence without making reference to the fact that it is, at first
;; appearance, a transitive verb.  This shows how Ehrlich's algorithim treats verbs
;; not labeled as bitransitive, transitive, etc.

(describe (assert act (build lex "throw")
                      agent (build lex "Derek")
                      object *baseball))

(M7! (ACT (M5 (LEX throw))) (AGENT (M6 (LEX Derek))) (OBJECT B1))

(M7!)

 CPU time : 0.00

*

;; Model the fact that ball is a basic level category.  This follows Ehrlich's case frames.
(describe (assert object1 (build lex "baseball")
                          rel ("ISA") object2
                          (build lex "basic ctgy")))

(M9! (OBJECT1 (M1 (LEX baseball))) (OBJECT2 (M8 (LEX basic ctgy))) (REL ISA))

(M9!)

 CPU time : 0.00

*
;; Include background knowledge about Derek.  Assume that his parent class is human and
;; that human is a basic level.
(describe (assert member (build lex "Derek")
                          class (build lex "human")))

(M11! (CLASS (M10 (LEX human))) (MEMBER (M6 (LEX Derek))))

(M11!)

 CPU time : 0.01

*
;; Model the fact that human is a basic level category.  This follows Ehrlich's case
;; frames
(describe (assert object1 (build lex "human")
                      rel ("ISA")
                      object2 (build lex "basic ctgy")))

(M12! (OBJECT1 (M10 (LEX human))) (OBJECT2 (M8 (LEX basic ctgy))) (REL ISA))

(M12!)
```

```
 CPU time : 0.00

*
;; The moment of truth...
^(
--> defn_verb 'throw)
(A (human) CAN THROW RESULT= NIL ENABLED BY= NIL)



 CPU time : 0.10

*
(resetnet)

Net reset - Relations and paths are still defined


 CPU time : 0.01

*
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;; Example Two - Demonstration of Ehrlich's verb algorithm with a verb that is labeled
;; as transitive.
;; The sentence:
;; "Derek threw the baseball."
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

;; Include background knowledge about a baseball.  Assume that ball is its parent class
;; and that ball is basic level.
(describe (assert subclass (build lex "baseball")
                  superclass (build lex "ball") ))

(M3! (SUBCLASS (M1 (LEX baseball))) (SUPERCLASS (M2 (LEX ball))))

(M3!)

 CPU time : 0.00

*
;; Make reference to the baseball without specifying which baseball it is.
(describe (assert member #baseball
                  class (build lex "baseball")))

(M4! (CLASS (M1 (LEX baseball))) (MEMBER B1))

(M4!)

 CPU time : 0.00

*
;; "Derek threw the baseball."
;; Represent the sentence without making reference to the fact that it is, at first
;; appearance, a transitive verb.  This shows how Ehrlich's algorithim treats verbs
;; not labeled as bitransitive, transitive, etc.
```

```
(describe (assert act (build lex "throw")
                      agent (build lex "Derek")
                      object *baseball))

(M7! (ACT (M5 (LEX throw))) (AGENT (M6 (LEX Derek))) (OBJECT B1))

(M7!)

 CPU time : 0.01

*

;; Model the fact that ball is a basic level category.  This follows Ehrlich's case frames.
(describe (assert object1 (build lex "baseball")
                      rel ("ISA")
                      object2 (build lex "basic ctgy")))

(M9! (OBJECT1 (M1 (LEX baseball))) (OBJECT2 (M8 (LEX basic ctgy))) (REL ISA))

(M9!)

 CPU time : 0.00

*
;; Include background knowledge about Derek.  Assume that his parent class is human and
;; that human is a basic level.
(describe (assert member (build lex "Derek")
                      class (build lex "human")))

(M11! (CLASS (M10 (LEX human))) (MEMBER (M6 (LEX Derek))))

(M11!)

 CPU time : 0.00

*
;; Model the fact that human is a basic level category.  This follows Ehrlich's case
;; frames
(describe (assert object1 (build lex "human")
                      rel ("ISA")
                      object2 (build lex "basic ctgy")))

(M12! (OBJECT1 (M10 (LEX human))) (OBJECT2 (M8 (LEX basic ctgy))) (REL ISA))

(M12!)

 CPU time : 0.01

*
;; Define throw as a transitive verb (at least in the context of the sentence)
(describe (assert object (build lex "throw")
                      property (build lex "transitive")))

(M14! (OBJECT (M5 (LEX throw))) (PROPERTY (M13 (LEX transitive))))
```

```
(M14!)

 CPU time : 0.00

*
;; The moment of truth...
^(
--> defn_verb 'throw)
(A (human) CAN THROW A (baseball) RESULT= NIL ENABLED BY= NIL)



 CPU time : 0.16

*
(resetnet)

Net reset - Relations and paths are still defined


 CPU time : 0.01

*

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;; Example Three - Demonstration of Ehrlich's verb algorithm with a verb that is
;; labeled as transitive when it is really bitransitive.
;; The sentence:
;; "Derek threw the baseball to Tino."
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

;; Include background knowledge about a baseball.  Assume that ball is its parent class
;; and that ball is basic level.
(describe (assert subclass (build lex "baseball")
                  superclass (build lex "ball") ))

(M3! (SUBCLASS (M1 (LEX baseball))) (SUPERCLASS (M2 (LEX ball))))

(M3!)

 CPU time : 0.01

*
;; Make reference to the baseball without specifying which baseball it is.
(describe (assert member #baseball
                  class (build lex "baseball")))

(M4! (CLASS (M1 (LEX baseball))) (MEMBER B1))

(M4!)

 CPU time : 0.00

*
;; "Derek threw the baseball."
;; Represent the sentence without making reference to the fact that it is, at first
```

```
;; appearance, a transitive verb.  This shows how Ehrlich's algorithim treats verbs
;; not labeled as bitransitive, transitive, etc.

(describe (assert act (build lex "throw")
                      agent (build lex "Derek")
                      object *baseball
                      indobj(build lex "Tino")))

(M8! (ACT (M5 (LEX throw))) (AGENT (M6 (LEX Derek))) (INDOBJ (M7 (LEX Tino)))
 (OBJECT B1))

(M8!)

 CPU time : 0.00

*
;; Model the fact that ball is a basic level category.  This follows Ehrlich's case frames.
(describe (assert object1 (build lex "baseball")
                      rel ("ISA")
                      object2 (build lex "basic ctgy")))

(M10! (OBJECT1 (M1 (LEX baseball))) (OBJECT2 (M9 (LEX basic ctgy))) (REL ISA))

(M10!)

 CPU time : 0.00

*
;; Include background knowledge about Derek.  Assume that his parent class is human and
;; that human is a basic level.
(describe (assert member (build lex "Derek")
                      class (build lex "human")))

(M12! (CLASS (M11 (LEX human))) (MEMBER (M6 (LEX Derek))))

(M12!)

 CPU time : 0.00

*
;; Include background knowledge about Tino.  Assume that his parent class is human and
;; that human is a basic level.
(describe (assert member (build lex "Tino")
                      class (build lex "human")))

(M13! (CLASS (M11 (LEX human))) (MEMBER (M7 (LEX Tino))))

(M13!)

 CPU time : 0.00

*
;; Model the fact that human is a basic level category.  This follows Ehrlich's case
;; frames
(describe (assert object1 (build lex "human")
                      rel ("ISA")
```

```
                              object2 (build lex "basic ctgy")))

(M14! (OBJECT1 (M11 (LEX human))) (OBJECT2 (M9 (LEX basic ctgy))) (REL ISA))

(M14!)

 CPU time : 0.01

*
;; Define throw as a transitive verb (at least in the context of the sentence)
(describe (assert object (build lex "throw")
                              property (build lex "transitive")))

(M16! (OBJECT (M5 (LEX throw))) (PROPERTY (M15 (LEX transitive))))

(M16!)

 CPU time : 0.00

*
;; The moment of truth...
^(
--> defn_verb 'throw)
(A (human) CAN THROW A (baseball) RESULT= NIL ENABLED BY= NIL)



 CPU time : 0.17

*
(resetnet)

Net reset - Relations and paths are still defined


 CPU time : 0.01

*
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;; Example Four - Demonstration of Ehrlich's verb algorithm with a verb that is
;; correctly labeled as bitransitive
;; The sentence:
;; "Derek threw the baseball to Tino."
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

;; Include background knowledge about a baseball.  Assume that ball is its parent class
;; and that ball is basic level.
(describe (assert subclass (build lex "baseball")
                              superclass (build lex "ball") ))

(M3! (SUBCLASS (M1 (LEX baseball))) (SUPERCLASS (M2 (LEX ball))))

(M3!)

 CPU time : 0.01
```

```
*
;; Make reference to the baseball without specifying which baseball it is.
(describe (assert member #baseball
                    class (build lex "baseball")))

(M4! (CLASS (M1 (LEX baseball))) (MEMBER B1))

(M4!)

 CPU time : 0.00

*
;; "Derek threw the baseball."
;; Represent the sentence without making reference to the fact that it is, at first
;; appearance, a transitive verb.  This shows how Ehrlich's algorithim treats verbs
;; not labeled as bitransitive, transitive, etc.

(describe (assert act (build lex "throw")
                    agent (build lex "Derek")
                    object *baseball
                    indobj(build lex "Tino")))

(M8! (ACT (M5 (LEX throw))) (AGENT (M6 (LEX Derek))) (INDOBJ (M7 (LEX Tino)))
 (OBJECT B1))

(M8!)

 CPU time : 0.00

*
;; Model the fact that ball is a basic level category.  This follows Ehrlich's case frames.
(describe (assert object1 (build lex "baseball")
                    rel ("ISA")
                    object2 (build lex "basic ctgy")))

(M10! (OBJECT1 (M1 (LEX baseball))) (OBJECT2 (M9 (LEX basic ctgy))) (REL ISA))

(M10!)

 CPU time : 0.01

*
;; Include background knowledge about Derek.  Assume that his parent class is human and
;; that human is a basic level.
(describe (assert member (build lex "Derek")
                    class (build lex "human")))

(M12! (CLASS (M11 (LEX human))) (MEMBER (M6 (LEX Derek))))

(M12!)

 CPU time : 0.00

*
;; Include background knowledge about Tino.  Assume that his parent class is human and
;; that human is a basic level.
```

```
(describe (assert member (build lex "Tino")
                  class (build lex "human")))

(M13! (CLASS (M11 (LEX human))) (MEMBER (M7 (LEX Tino))))

(M13!)

 CPU time : 0.00

*
;; Model the fact that human is a basic level category.  This follows Ehrlich's case
;; frames
(describe (assert object1 (build lex "human")
                  rel ("ISA")
                  object2 (build lex "basic ctgy")))

(M14! (OBJECT1 (M11 (LEX human))) (OBJECT2 (M9 (LEX basic ctgy))) (REL ISA))

(M14!)

 CPU time : 0.01

*
;; Define throw as a transitive verb (at least in the context of the sentence)
(describe (assert object (build lex "throw")
                  property (build lex "bitransitive")))

(M16! (OBJECT (M5 (LEX throw))) (PROPERTY (M15 (LEX bitransitive))))

(M16!)

 CPU time : 0.00

*
;; The moment of truth...
^(
--> defn_verb 'throw)
(A (human) CAN THROW A (baseball) TO A (human) RESULT= NIL ENABLED BY= NIL)



 CPU time : 0.31

*
(resetnet)

Net reset - Relations and paths are still defined


 CPU time : 0.01

*
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;; Example Five - Demonstration of Ehrlich's verb algorithm with multiple instances
;; of the same verb.
;; The sentences are:
```

```
;; "Derek threw the baseball to Tino."
;; "Bobo throws the rubber ball."
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

;; Include background knowledge about a baseball.  Assume that ball is its parent class
;; and that ball is basic level.
(describe (assert subclass (build lex "baseball")
                  superclass (build lex "ball") ))

(M3! (SUBCLASS (M1 (LEX baseball))) (SUPERCLASS (M2 (LEX ball))))

(M3!)

 CPU time : 0.01

*
;; Make reference to the baseball without specifying which baseball it is.
(describe (assert member #baseball
                  class (build lex "baseball")))

(M4! (CLASS (M1 (LEX baseball))) (MEMBER B1))

(M4!)

 CPU time : 0.00

*
;; "Derek threw the baseball."
;; Represent the sentence without making reference to the fact that it is, at first
;; appearance, a transitive verb.  This shows how Ehrlich's algorithim treats verbs
;; not labeled as bitransitive, transitive, etc.

(describe (assert act (build lex "throw")
                  agent (build lex "Derek")
                  object *baseball
                  indobj(build lex "Tino")))

(M8! (ACT (M5 (LEX throw))) (AGENT (M6 (LEX Derek))) (INDOBJ (M7 (LEX Tino)))
 (OBJECT B1))

(M8!)

 CPU time : 0.00

*
;; Model the fact that ball is a basic level category.  This follows Ehrlich's case frames.
(describe (assert object1 (build lex "baseball")
                  rel ("ISA")
                  object2 (build lex "basic ctgy")))

(M10! (OBJECT1 (M1 (LEX baseball))) (OBJECT2 (M9 (LEX basic ctgy))) (REL ISA))

(M10!)

 CPU time : 0.01
```

```
*
;; Include background knowledge about Derek.  Assume that his parent class is human and
;; that human is a basic level.
(describe (assert member (build lex "Derek")
                         class (build lex "human")))

(M12! (CLASS (M11 (LEX human))) (MEMBER (M6 (LEX Derek))))

(M12!)

 CPU time : 0.00

*
;; Include background knowledge about Tino.  Assume that his parent class is human and
;; that human is a basic level.
(describe (assert member (build lex "Tino")
                         class (build lex "human")))

(M13! (CLASS (M11 (LEX human))) (MEMBER (M7 (LEX Tino))))

(M13!)

 CPU time : 0.00

*
;; Model the fact that human is a basic level category.  This follows Ehrlich's case
;; frames
(describe (assert object1 (build lex "human")
                         rel ("ISA")
                         object2 (build lex "basic ctgy")))

(M14! (OBJECT1 (M11 (LEX human))) (OBJECT2 (M9 (LEX basic ctgy))) (REL ISA))

(M14!)

 CPU time : 0.00

*
;; Define throw as a transitive verb (at least in the context of the sentence)
(describe (assert object (build lex "throw")
                         property (build lex "bitransitive")))

(M16! (OBJECT (M5 (LEX throw))) (PROPERTY (M15 (LEX bitransitive))))

(M16!)

 CPU time : 0.00

*
;; "Bobo throws the rubber ball"
;; Here assume that Bobo is a chimp.  Assume that chimp is a basic level category as well
;; as rubber ball.
;; Note that the sentence uses throw in a transitive sense.  Will the algoritm make a distinction
;; between the type of throwing a human can do and that which a chimp can do?

;; Include background knowledge about a rubber ball.  Assume that ball is its parent class.
```

```
(describe (assert subclass (build lex "rubber ball")
                        superclass (build lex "ball") ))

(M18! (SUBCLASS (M17 (LEX rubber ball))) (SUPERCLASS (M2 (LEX ball))))

(M18!)

 CPU time : 0.00

*
;; Make reference to the baseball without specifying which baseball it is.
(describe (assert member #rubberBall
                        class (build lex "rubber ball")))

(M19! (CLASS (M17 (LEX rubber ball))) (MEMBER B2))

(M19!)

 CPU time : 0.00

*
;; Model the action.
(describe (assert act (build lex "throw")
                        agent (build lex "Bobo")
                        object *rubberBall
                        ))

(M21! (ACT (M5 (LEX throw))) (AGENT (M20 (LEX Bobo))) (OBJECT B2))

(M21!)

 CPU time : 0.00

*
;; Model the fact that rubberball is a basic level category.
;; This follows Ehrlich's case frames.
(describe (assert object1 (build lex "rubber ball")
                        rel ("ISA")
                        object2 (build lex "basic ctgy")))

(M22! (OBJECT1 (M17 (LEX rubber ball))) (OBJECT2 (M9 (LEX basic ctgy)))
 (REL ISA))

(M22!)

 CPU time : 0.00

*
;; Model that Bobo is a chimp.
(describe (assert member (build lex "Bobo")
                        class (build lex "chimp")))

(M24! (CLASS (M23 (LEX chimp))) (MEMBER (M20 (LEX Bobo))))

(M24!)
```

```
 CPU time : 0.00

*
;; Model that chimp is a basic level category.
(describe (assert object1 (build lex "chimp")
                  rel ("ISA")
                  object2 (build lex "basic ctgy")))

(M25! (OBJECT1 (M23 (LEX chimp))) (OBJECT2 (M9 (LEX basic ctgy))) (REL ISA))

(M25!)

 CPU time : 0.00

*
;; The moment of truth...
^(
--> defn_verb 'throw)
(A (human chimp) CAN THROW A (baseball rubber ball) TO A (human) RESULT= NIL
 ENABLED BY= NIL)



 CPU time : 0.33

*
(resetnet)

Net reset - Relations and paths are still defined


 CPU time : 0.01

*
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;; Example Six - Demonstration of Ehrlich's verb algorithm with a verb used multiple
;; times in both transitive and bitransitive forms.  However, the transitive and
;; bitransitive lexes are set for the verb each time!  Will Ehrlich's algorithm
;; differentiate between the two?
;; The sentences are:
;; "Derek threw the baseball to Tino."
;; "Bobo throws the rubber ball."
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

;; Include background knowledge about a baseball.  Assume that ball is its parent class
;; and that ball is basic level.
(describe (assert subclass (build lex "baseball")
                  superclass (build lex "ball") ))

(M3! (SUBCLASS (M1 (LEX baseball))) (SUPERCLASS (M2 (LEX ball))))

(M3!)

 CPU time : 0.00

*
```

```
;; Make reference to the baseball without specifying which baseball it is.
(describe (assert member #baseball
                    class (build lex "baseball")))

(M4! (CLASS (M1 (LEX baseball))) (MEMBER B1))

(M4!)

 CPU time : 0.01

*
;; "Derek threw the baseball."
;; Represent the sentence without making reference to the fact that it is, at first
;; appearance, a transitive verb.  This shows how Ehrlich's algorithim treats verbs
;; not labeled as bitransitive, transitive, etc.

(describe (assert act (build lex "throw")
                    agent (build lex "Derek")
                    object *baseball
                    indobj(build lex "Tino")))

(M8! (ACT (M5 (LEX throw))) (AGENT (M6 (LEX Derek))) (INDOBJ (M7 (LEX Tino)))
 (OBJECT B1))

(M8!)

 CPU time : 0.00

*
;; Model the fact that ball is a basic level category.  This follows Ehrlich's case frames.
(describe (assert object1 (build lex "baseball")
                    rel ("ISA")
                    object2 (build lex "basic ctgy")))

(M10! (OBJECT1 (M1 (LEX baseball))) (OBJECT2 (M9 (LEX basic ctgy))) (REL ISA))

(M10!)

 CPU time : 0.00

*
;; Include background knowledge about Derek.  Assume that his parent class is human and
;; that human is a basic level.
(describe (assert member (build lex "Derek")
                    class (build lex "human")))

(M12! (CLASS (M11 (LEX human))) (MEMBER (M6 (LEX Derek))))

(M12!)

 CPU time : 0.00

*
;; Include background knowledge about Tino.  Assume that his parent class is human and
;; that human is a basic level.
(describe (assert member (build lex "Tino")
```

```
                    class (build lex "human")))

(M13! (CLASS (M11 (LEX human))) (MEMBER (M7 (LEX Tino))))

(M13!)

 CPU time : 0.00

*
;; Model the fact that human is a basic level category.  This follows Ehrlich's case
;; frames
(describe (assert object1 (build lex "human")
                    rel ("ISA")
                    object2 (build lex "basic ctgy")))

(M14! (OBJECT1 (M11 (LEX human))) (OBJECT2 (M9 (LEX basic ctgy))) (REL ISA))

(M14!)

 CPU time : 0.01

*
;; Define throw as a transitive verb (at least in the context of the sentence)
(describe (assert object (build lex "throw")
                    property (build lex "bitransitive")))

(M16! (OBJECT (M5 (LEX throw))) (PROPERTY (M15 (LEX bitransitive))))

(M16!)

 CPU time : 0.00

*
;; "Bobo throws the rubber ball"
;; Here assume that Bobo is a chimp.  Assume that chimp is a basic level category as well
;; as rubber ball.
;; Note that the sentence uses throw in a transitive sense.  Will the algoritm make a distinction
;; between the type of throwing a human can do and that which a chimp can do?

;; Include background knowledge about a rubber ball.  Assume that ball is its parent class.
(describe (assert subclass (build lex "rubber ball")
                    superclass (build lex "ball") ))

(M18! (SUBCLASS (M17 (LEX rubber ball))) (SUPERCLASS (M2 (LEX ball))))

(M18!)

 CPU time : 0.00

*
;; Make reference to the baseball without specifying which baseball it is.
(describe (assert member #rubberBall
                    class (build lex "rubber ball")))

(M19! (CLASS (M17 (LEX rubber ball))) (MEMBER B2))
```

```
(M19!)

 CPU time : 0.01

*
;; Model the action.
(describe (assert act (build lex "throw")
                      agent (build lex "Bobo")
                      object *rubberBall
                      ))

(M21! (ACT (M5 (LEX throw))) (AGENT (M20 (LEX Bobo))) (OBJECT B2))

(M21!)

 CPU time : 0.00

*
;; Define throw as a transitive verb.  Now it has been defined in two ways.  Will this affect
;; Ehrlich's algorithm?
(describe (assert object (build lex "throw")
                      property (build lex "transitive")))

(M23! (OBJECT (M5 (LEX throw))) (PROPERTY (M22 (LEX transitive))))

(M23!)

 CPU time : 0.00

*
;; Model the fact that rubberball is a basic level category.
;; This follows Ehrlich's case frames.
(describe (assert object1 (build lex "rubber ball")
                      rel ("ISA")
                      object2 (build lex "basic ctgy")))

(M24! (OBJECT1 (M17 (LEX rubber ball))) (OBJECT2 (M9 (LEX basic ctgy)))
 (REL ISA))

(M24!)

 CPU time : 0.01

*
;; Model that Bobo is a chimp.
(describe (assert member (build lex "Bobo")
                      class (build lex "chimp")))

(M26! (CLASS (M25 (LEX chimp))) (MEMBER (M20 (LEX Bobo))))

(M26!)

 CPU time : 0.00

*
;; Model that chimp is a basic level category.
```

```
(describe (assert object1 (build lex "chimp")
                  rel ("ISA")
                  object2 (build lex "basic ctgy")))

(M27! (OBJECT1 (M25 (LEX chimp))) (OBJECT2 (M9 (LEX basic ctgy))) (REL ISA))

(M27!)

 CPU time : 0.00

*
;; The moment of truth...
^(
--> defn_verb 'throw)
(A (human chimp) CAN THROW A (baseball rubber ball) TO A (human) RESULT= NIL
 ENABLED BY= NIL)



 CPU time : 0.31

*
(resetnet)

Net reset - Relations and paths are still defined


 CPU time : 0.01

*
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;; Example Seven - Demonstrate Ehrlich's algorithm with a verb that has a consequence
;; defined for it.
;; The sentences are:
;; "Derek threw the baseball to Tino."
;; "Bobo throws the rubber ball."
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

;; Include background knowledge about a baseball.  Assume that ball is its parent class
;; and that ball is basic level.
(describe (assert subclass (build lex "baseball")
                  superclass (build lex "ball") ))

(M3! (SUBCLASS (M1 (LEX baseball))) (SUPERCLASS (M2 (LEX ball))))

(M3!)

 CPU time : 0.00

*
;; Make reference to the baseball without specifying which baseball it is.
(describe (assert member #baseball
                  class (build lex "baseball")))

(M4! (CLASS (M1 (LEX baseball))) (MEMBER B1))
```

```
(M4!)

 CPU time : 0.01

*
;; "Derek threw the baseball."
;; Represent the sentence without making reference to the fact that it is, at first
;; appearance, a transitive verb.  This shows how Ehrlich's algorithim treats verbs
;; not labeled as bitransitive, transitive, etc.

(describe (assert act (build lex "throw")
                      agent (build lex "Derek")
                      object *baseball
                      indobj(build lex "Tino")))

(M8! (ACT (M5 (LEX throw))) (AGENT (M6 (LEX Derek))) (INDOBJ (M7 (LEX Tino)))
 (OBJECT B1))

(M8!)

 CPU time : 0.00

*
;; Model the fact that ball is a basic level category.  This follows Ehrlich's case frames.
(describe (assert object1 (build lex "baseball")
                      rel ("ISA")
                      object2 (build lex "basic ctgy")))

(M10! (OBJECT1 (M1 (LEX baseball))) (OBJECT2 (M9 (LEX basic ctgy))) (REL ISA))

(M10!)

 CPU time : 0.01

*
;; Include background knowledge about Derek.  Assume that his parent class is human and
;; that human is a basic level.
(describe (assert member (build lex "Derek")
                      class (build lex "human")))

(M12! (CLASS (M11 (LEX human))) (MEMBER (M6 (LEX Derek))))

(M12!)

 CPU time : 0.01

*
;; Include background knowledge about Tino.  Assume that his parent class is human and
;; that human is a basic level.
(describe (assert member (build lex "Tino")
                      class (build lex "human")))

(M13! (CLASS (M11 (LEX human))) (MEMBER (M7 (LEX Tino))))

(M13!)
```

```
 CPU time : 0.00

*
;; Model the fact that human is a basic level category.  This follows Ehrlich's case
;; frames
(describe (assert object1 (build lex "human")
                          rel ("ISA")
                          object2 (build lex "basic ctgy")))

(M14! (OBJECT1 (M11 (LEX human))) (OBJECT2 (M9 (LEX basic ctgy))) (REL ISA))

(M14!)

 CPU time : 0.00

*
;; Define throw as a transitive verb (at least in the context of the sentence)
(describe (assert object (build lex "throw")
                          property (build lex "bitransitive")))

(M16! (OBJECT (M5 (LEX throw))) (PROPERTY (M15 (LEX bitransitive))))

(M16!)

 CPU time : 0.00

*
;; "Bobo throws the rubber ball"
;; Here assume that Bobo is a chimp.  Assume that chimp is a basic level category as well
;; as rubber ball.
;; Note that the sentence uses throw in a transitive sense.  Will the algoritm make a distinction
;; between the type of throwing a human can do and that which a chimp can do?

;; Include background knowledge about a rubber ball.  Assume that ball is its parent class.
(describe (assert subclass (build lex "rubber ball")
                          superclass (build lex "ball") ))

(M18! (SUBCLASS (M17 (LEX rubber ball))) (SUPERCLASS (M2 (LEX ball))))

(M18!)

 CPU time : 0.00

*
;; Make reference to the baseball without specifying which baseball it is.
(describe (assert member #rubberBall
                          class (build lex "rubber ball")))

(M19! (CLASS (M17 (LEX rubber ball))) (MEMBER B2))

(M19!)

 CPU time : 0.01

*
;; Model the action.
```

```
(describe (assert act (build lex "throw")
                      agent (build lex "Bobo")
                      object *rubberBall
                      ))

(M21! (ACT (M5 (LEX throw))) (AGENT (M20 (LEX Bobo))) (OBJECT B2))

(M21!)

 CPU time : 0.00

*
;; Define throw as a transitive verb.  Now it has been defined in two ways.  Will this affect
;; Ehrlich's algorithm?
(describe (assert object (build lex "throw")
                      property (build lex "transitive")))

(M23! (OBJECT (M5 (LEX throw))) (PROPERTY (M22 (LEX transitive))))

(M23!)

 CPU time : 0.02

*
;; Model the fact that rubberball is a basic level category.
;; This follows Ehrlich's case frames.
(describe (assert object1 (build lex "rubber ball")
                      rel ("ISA")
                      object2 (build lex "basic ctgy")))

(M24! (OBJECT1 (M17 (LEX rubber ball))) (OBJECT2 (M9 (LEX basic ctgy)))
 (REL ISA))

(M24!)

 CPU time : 0.00

*
;; Model that Bobo is a chimp.
(describe (assert member (build lex "Bobo")
                      class (build lex "chimp")))

(M26! (CLASS (M25 (LEX chimp))) (MEMBER (M20 (LEX Bobo))))

(M26!)

 CPU time : 0.00

*
;; Model that chimp is a basic level category.
(describe (assert object1 (build lex "chimp")
                      rel ("ISA")
                      object2 (build lex "basic ctgy")))

(M27! (OBJECT1 (M25 (LEX chimp))) (OBJECT2 (M9 (LEX basic ctgy))) (REL ISA))
```

```
(M27!)

 CPU time : 0.00

*
;; Create a path-based inference we will need for subclasses
;; Notice the results this has on the verb algorithms output!
(define-path class
          (compose class
                    (kstar (compose subclass- ! superclass))))
CLASS implied by the path (COMPOSE CLASS
                    (KSTAR (COMPOSE SUBCLASS- ! SUPERCLASS)))
CLASS- implied by the path (COMPOSE (KSTAR (COMPOSE SUPERCLASS- ! SUBCLASS))
                    CLASS-)


 CPU time : 0.01

*
;; Create a consequence (in Ehrlichs terms) for throwing.  This consequence is what
;; Ehrlich would have considered a life-rule2
;; "If a person x throws a ball then he lofts it in the air

(describe (assert forall ($h $b)
                    &ant((build member *h
                              class (build lex "human"))
                       (build member *b
                              class (build lex "ball"))
                       (build agent *h
                              act (build lex "throw")
                              object *b))
                    cq (build agent *h
                              act (build lex "loft")
                              object *b
                              location (build lex "through the air")
                              )
                    kn_cat ("life-rule.2")))

(M30! (FORALL V2 V1)
 (&ANT (P3 (ACT (M5 (LEX throw))) (AGENT V1) (OBJECT V2))
  (P2 (CLASS (M2 (LEX ball))) (MEMBER V2))
  (P1 (CLASS (M11 (LEX human))) (MEMBER V1)))
 (CQ
  (P4 (ACT (M28 (LEX loft))) (AGENT V1) (LOCATION (M29 (LEX through the air)))
   (OBJECT V2)))
 (KN_CAT life-rule.2))

(M30!)

 CPU time : 0.01

*
;; Proof that the rule is in working order
(describe (deduce agent $whoever act (build lex "loft")))

(M35! (ACT (M28 (LEX loft))) (AGENT (M6 (LEX Derek))))
```

```
(M35!)

 CPU time : 0.02

*

^(
--> defn_verb 'throw)
(A (human chimp) CAN THROW A (baseball ball rubber ball) TO A (human) RESULT=
 (P4) ENABLED BY= NIL)



 CPU time : 0.43

*
(resetnet)

Net reset - Relations and paths are still defined


 CPU time : 0.01

*



End of /home/csgrad/sood/cse663/demos/throw.demo demonstration.


 CPU time : 2.30

*
```

8.3.2 Results with my improved version (with trace turned on)

```
============================================================
Starting image `/util/acl5.0.1/composer'
  with no arguments
  in directory `/home/csgrad/sood/cse663/demos/'
  on machine `localhost'.

Allegro CL Enterprise Edition 5.0.1 [SPARC] (8/20/99 10:07)
Copyright (C) 1985-1999, Franz Inc., Berkeley, CA, USA.  All Rights Reserved.
;; 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).
```

```
USER(1): :ld /projects/snwiz/bin/sneps
; Fast loading /projects/snwiz/bin/sneps.fasl
Loading system SNePS...10% 20% 30% 40% 50% 60% 70% 80% 90% 100%
SNePS-2.5 [PL:1 1999/08/19 16:38:25] loaded.
Type `(sneps)' or `(snepslog)' to get started.
USER(2): (sneps)


   Welcome to SNePS-2.5 [PL:1 1999/08/19 16:38:25]

Copyright (C) 1984--1999 by Research Foundation of
State University of New York. SNePS comes with ABSOLUTELY NO WARRANTY!
Type `(copyright)' for detailed copyright information.
Type `(demo)' for a list of example applications.

   5/6/2002 2:46:05

* (demo "throw_well.demo")

File /home/csgrad/sood/cse663/demos/throw_well.demo is now the source of input.


 CPU time : 0.01

* ;;; Reset the network
(resetnet t)

Net reset


 CPU time : 0.01

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



 CPU time : 0.00

*
;;; Introduce Marc's relations
(intext "rels")
File rels is now the source of input.


 CPU time : 0.00

* _ACT is already defined.
_ACTION is already defined.
_EFFECT is already defined.
_OBJECT1 is already defined.
_OBJECT2 is already defined.
```

(A1 A2 A3 A4 ACT ACTION AFTER AGENT ANTONYM ASSOCIATED BEFORE CAUSE CLASS
DIRECTION EFFECT EQUIV ETIME FROM IN INDOBJ INSTR INTO LEX LOCATION KN_CAT
MANNER MEMBER MEMBERS MODE OBJECT OBJECTS OBJECT1 OBJECTS1 OBJECT2 ON ONTO
PART PLACE POSSESSOR PROPER-NAME PROPERTY PURPOSE REL SKF STIME SUBCLASS
SUPERCLASS SYNONYM TIME TO WHOLE)

CPU time : 0.06

*

End of file rels

CPU time : 0.07

*
;;; Compose paths
(intext "paths")
File paths is now the source of input.

CPU time : 0.00

*
BEFORE implied by the path (COMPOSE BEFORE (KSTAR (COMPOSE AFTER- ! BEFORE)))
BEFORE- implied by the path (COMPOSE (KSTAR (COMPOSE BEFORE- ! AFTER)) BEFORE-)

CPU time : 0.01

*
AFTER implied by the path (COMPOSE AFTER (KSTAR (COMPOSE BEFORE- ! AFTER)))
AFTER- implied by the path (COMPOSE (KSTAR (COMPOSE AFTER- ! BEFORE)) AFTER-)

CPU time : 0.00

*
SUB1 implied by the path (COMPOSE OBJECT1- SUPERCLASS- ! SUBCLASS SUPERCLASS-
! SUBCLASS)
SUB1- implied by the path (COMPOSE SUBCLASS- ! SUPERCLASS SUBCLASS- !
SUPERCLASS OBJECT1)

CPU time : 0.00

*
SUPER1 implied by the path (COMPOSE SUPERCLASS SUBCLASS- ! SUPERCLASS OBJECT1-
! OBJECT2)
SUPER1- implied by the path (COMPOSE OBJECT2- ! OBJECT1 SUPERCLASS- ! SUBCLASS
SUPERCLASS-)

CPU time : 0.00

*

```
SUPERCLASS implied by the path (OR SUPERCLASS SUPER1)
SUPERCLASS- implied by the path (OR SUPERCLASS- SUPER1-)


 CPU time : 0.00

*

End of file paths


 CPU time : 0.01

*
;;; Load Noun/Verb Algorithm
^(
--> load "fast_verb_fixed_well")
; Fast loading /home/csgrad/sood/cse663/demos/fast_verb_fixed_well.fasl
T



 CPU time : 0.01

*
;;; turn on defn_verb trace
^(
--> setq verb_trace t)
T



 CPU time : 0.00

*
;; Define the relations necessary
(define lex act agent member class object1 rel object2 property object indobj cause effect kn_cat)
_LEX is already defined.
_ACT is already defined.
_AGENT is already defined.
_MEMBER is already defined.
_CLASS is already defined.
_OBJECT1 is already defined.
_REL is already defined.
_OBJECT2 is already defined.
_PROPERTY is already defined.
_OBJECT is already defined.
_INDOBJ is already defined.
_CAUSE is already defined.
_EFFECT is already defined.
_KN_CAT is already defined.

(LEX ACT AGENT MEMBER CLASS OBJECT1 REL OBJECT2 PROPERTY OBJECT INDOBJ CAUSE
 EFFECT KN_CAT)

 CPU time : 0.00
```

```
*

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;; Example One - Demonstration of Ehrlich's verb algorithm with a verb not labeled as
;; bitransitive, transitive, etc.
;; The sentence:
;; "Derek threw the baseball."
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

;; Include background knowledge about a baseball.  Assume that ball is its parent class
;; and that ball is basic level.
(describe (assert subclass (build lex "baseball")
                  superclass (build lex "ball") ))

(M3! (SUBCLASS (M1 (LEX baseball))) (SUPERCLASS (M2 (LEX ball))))

(M3!)

 CPU time : 0.00

*
;; Make reference to the baseball without specifying which baseball it is.
(describe (assert member #baseball
                  class (build lex "baseball")))

(M4! (CLASS (M1 (LEX baseball))) (MEMBER B1))

(M4!)

 CPU time : 0.00

*
;; "Derek threw the baseball."
;; Represent the sentence without making reference to the fact that it is, at first
;; appearance, a transitive verb.  This shows how Ehrlich's algorithim treats verbs
;; not labeled as bitransitive, transitive, etc.

(describe (assert act (build lex "throw")
                  agent (build lex "Derek")
                  object *baseball))

(M7! (ACT (M5 (LEX throw))) (AGENT (M6 (LEX Derek))) (OBJECT B1))

(M7!)

 CPU time : 0.00

*

;; Model the fact that ball is a basic level category.  This follows Ehrlich's case frames.
(describe (assert object1 (build lex "baseball")
                         rel ("ISA") object2
                         (build lex "basic ctgy")))

(M9! (OBJECT1 (M1 (LEX baseball))) (OBJECT2 (M8 (LEX basic ctgy))) (REL ISA))
```

```
(M9!)

 CPU time : 0.01

*
;; Include background knowledge about Derek.  Assume that his parent class is human and
;; that human is a basic level.
(describe (assert member (build lex "Derek")
                  class (build lex "human")))

(M11! (CLASS (M10 (LEX human))) (MEMBER (M6 (LEX Derek))))

(M11!)

 CPU time : 0.00

*
;; Model the fact that human is a basic level category.  This follows Ehrlich's case
;; frames
(describe (assert object1 (build lex "human")
              rel ("ISA")
              object2 (build lex "basic ctgy")))

(M12! (OBJECT1 (M10 (LEX human))) (OBJECT2 (M8 (LEX basic ctgy))) (REL ISA))

(M12!)

 CPU time : 0.00

*
;; The moment of truth...
^(
--> defn_verb 'throw)
((A (human) CAN VERB A (baseball) RESULT= NIL ENABLED BY= NIL)
 ((SUBJECT FOUND BY BASE_CAT_SUBJ) (OBJECT FOUND BY BASE_CAT_OBJ)
  (INDIRECT OBJECT FOUND BY SOME_CAT_INDOBJ) (NO CAUSES FOUND)
  (NO EFFECTS FOUND)))



 CPU time : 0.15

*
(resetnet)

Net reset - Relations and paths are still defined


 CPU time : 0.01

*
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;; Example Two - Demonstration of Ehrlich's verb algorithm with a verb that is labeled
;; as transitive.
;; The sentence:
```

```
;; "Derek threw the baseball."
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

;; Include background knowledge about a baseball.  Assume that ball is its parent class
;; and that ball is basic level.
(describe (assert subclass (build lex "baseball")
                  superclass (build lex "ball") ))

(M3! (SUBCLASS (M1 (LEX baseball))) (SUPERCLASS (M2 (LEX ball))))

(M3!)

 CPU time : 0.00

*
;; Make reference to the baseball without specifying which baseball it is.
(describe (assert member #baseball
                  class (build lex "baseball")))

(M4! (CLASS (M1 (LEX baseball))) (MEMBER B1))

(M4!)

 CPU time : 0.00

*
;; "Derek threw the baseball."
;; Represent the sentence without making reference to the fact that it is, at first
;; appearance, a transitive verb.  This shows how Ehrlich's algorithim treats verbs
;; not labeled as bitransitive, transitive, etc.

(describe (assert act (build lex "throw")
                  agent (build lex "Derek")
                  object *baseball))

(M7! (ACT (M5 (LEX throw))) (AGENT (M6 (LEX Derek))) (OBJECT B1))

(M7!)

 CPU time : 0.00

*

;; Model the fact that ball is a basic level category.  This follows Ehrlich's case frames.
(describe (assert object1 (build lex "baseball")
                  rel ("ISA")
                  object2 (build lex "basic ctgy")))

(M9! (OBJECT1 (M1 (LEX baseball))) (OBJECT2 (M8 (LEX basic ctgy))) (REL ISA))

(M9!)

 CPU time : 0.01

*
;; Include background knowledge about Derek.  Assume that his parent class is human and
```

```
;; that human is a basic level.
(describe (assert member (build lex "Derek")
                         class (build lex "human")))

(M11! (CLASS (M10 (LEX human))) (MEMBER (M6 (LEX Derek))))

(M11!)

 CPU time : 0.00

*
;; Model the fact that human is a basic level category.  This follows Ehrlich's case
;; frames
(describe (assert object1 (build lex "human")
                         rel ("ISA")
                         object2 (build lex "basic ctgy")))

(M12! (OBJECT1 (M10 (LEX human))) (OBJECT2 (M8 (LEX basic ctgy))) (REL ISA))

(M12!)

 CPU time : 0.00

*
;; Define throw as a transitive verb (at least in the context of the sentence)
(describe (assert object (build lex "throw")
                         property (build lex "transitive")))

(M14! (OBJECT (M5 (LEX throw))) (PROPERTY (M13 (LEX transitive))))

(M14!)

 CPU time : 0.00

*
;; The moment of truth...
^(
--> defn_verb 'throw)
((A (human) CAN VERB A (baseball) RESULT= NIL ENABLED BY= NIL)
 ((SUBJECT FOUND BY BASE_CAT_SUBJ) (OBJECT FOUND BY BASE_CAT_OBJ)
  (INDIRECT OBJECT FOUND BY SOME_CAT_INDOBJ) (NO CAUSES FOUND)
  (NO EFFECTS FOUND)))



 CPU time : 0.15

*
(resetnet)

Net reset - Relations and paths are still defined


 CPU time : 0.01

*
```

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;; Example Three - Demonstration of Ehrlich's verb algorithm with a verb that is
;; labeled as transitive when it is really bitransitive.
;; The sentence:
;; "Derek threw the baseball to Tino."
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

;; Include background knowledge about a baseball.  Assume that ball is its parent class
;; and that ball is basic level.
(describe (assert subclass (build lex "baseball")
                  superclass (build lex "ball") ))

(M3! (SUBCLASS (M1 (LEX baseball))) (SUPERCLASS (M2 (LEX ball))))

(M3!)

 CPU time : 0.01

*
;; Make reference to the baseball without specifying which baseball it is.
(describe (assert member #baseball
                  class (build lex "baseball")))

(M4! (CLASS (M1 (LEX baseball))) (MEMBER B1))

(M4!)

 CPU time : 0.00

*
;; "Derek threw the baseball."
;; Represent the sentence without making reference to the fact that it is, at first
;; appearance, a transitive verb.  This shows how Ehrlich's algorithim treats verbs
;; not labeled as bitransitive, transitive, etc.

(describe (assert act (build lex "throw")
                  agent (build lex "Derek")
                  object *baseball
                  indobj(build lex "Tino")))

(M8! (ACT (M5 (LEX throw))) (AGENT (M6 (LEX Derek))) (INDOBJ (M7 (LEX Tino)))
 (OBJECT B1))

(M8!)

 CPU time : 0.01

*
;; Model the fact that ball is a basic level category.  This follows Ehrlich's case frames.
(describe (assert object1 (build lex "baseball")
                  rel ("ISA")
                  object2 (build lex "basic ctgy")))

(M10! (OBJECT1 (M1 (LEX baseball))) (OBJECT2 (M9 (LEX basic ctgy))) (REL ISA))
```

```
(M10!)

 CPU time : 0.00

*
;; Include background knowledge about Derek.  Assume that his parent class is human and
;; that human is a basic level.
(describe (assert member (build lex "Derek")
                         class (build lex "human")))

(M12! (CLASS (M11 (LEX human))) (MEMBER (M6 (LEX Derek))))

(M12!)

 CPU time : 0.00

*
;; Include background knowledge about Tino.  Assume that his parent class is human and
;; that human is a basic level.
(describe (assert member (build lex "Tino")
                         class (build lex "human")))

(M13! (CLASS (M11 (LEX human))) (MEMBER (M7 (LEX Tino))))

(M13!)

 CPU time : 0.01

*
;; Model the fact that human is a basic level category.  This follows Ehrlich's case
;; frames
(describe (assert object1 (build lex "human")
                         rel ("ISA")
                         object2 (build lex "basic ctgy")))

(M14! (OBJECT1 (M11 (LEX human))) (OBJECT2 (M9 (LEX basic ctgy))) (REL ISA))

(M14!)

 CPU time : 0.00

*
;; Define throw as a transitive verb (at least in the context of the sentence)
(describe (assert object (build lex "throw")
                         property (build lex "transitive")))

(M16! (OBJECT (M5 (LEX throw))) (PROPERTY (M15 (LEX transitive))))

(M16!)

 CPU time : 0.00

*
;; The moment of truth...
^(
--> defn_verb 'throw)
```

```
((A (human) CAN VERB A (baseball) TO A (human) RESULT= NIL ENABLED BY= NIL)
 ((SUBJECT FOUND BY BASE_CAT_SUBJ) (OBJECT FOUND BY BASE_CAT_OBJ)
  (INDIRECT OBJECT FOUND BY BASE_CAT_INDOBJ) (NO CAUSES FOUND)
  (NO EFFECTS FOUND)))



 CPU time : 0.18

*
(resetnet)

Net reset - Relations and paths are still defined


 CPU time : 0.02

*
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;; Example Four - Demonstration of Ehrlich's verb algorithm with a verb that is
;; correctly labeled as bitransitive
;; The sentence:
;; "Derek threw the baseball to Tino."
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

;; Include background knowledge about a baseball.  Assume that ball is its parent class
;; and that ball is basic level.
(describe (assert subclass (build lex "baseball")
                  superclass (build lex "ball") ))

(M3! (SUBCLASS (M1 (LEX baseball))) (SUPERCLASS (M2 (LEX ball))))

(M3!)

 CPU time : 0.00

*
;; Make reference to the baseball without specifying which baseball it is.
(describe (assert member #baseball
                  class (build lex "baseball")))

(M4! (CLASS (M1 (LEX baseball))) (MEMBER B1))

(M4!)

 CPU time : 0.00

*
;; "Derek threw the baseball."
;; Represent the sentence without making reference to the fact that it is, at first
;; appearance, a transitive verb.  This shows how Ehrlich's algorithim treats verbs
;; not labeled as bitransitive, transitive, etc.

(describe (assert act (build lex "throw")
                  agent (build lex "Derek")
                  object *baseball
```

```
                    indobj(build lex "Tino")))

(M8! (ACT (M5 (LEX throw))) (AGENT (M6 (LEX Derek))) (INDOBJ (M7 (LEX Tino)))
 (OBJECT B1))

(M8!)

 CPU time : 0.00

*
;; Model the fact that ball is a basic level category.  This follows Ehrlich's case frames.
(describe (assert object1 (build lex "baseball")
                    rel ("ISA")
                    object2 (build lex "basic ctgy")))

(M10! (OBJECT1 (M1 (LEX baseball))) (OBJECT2 (M9 (LEX basic ctgy))) (REL ISA))

(M10!)

 CPU time : 0.00

*
;; Include background knowledge about Derek.  Assume that his parent class is human and
;; that human is a basic level.
(describe (assert member (build lex "Derek")
                    class (build lex "human")))

(M12! (CLASS (M11 (LEX human))) (MEMBER (M6 (LEX Derek))))

(M12!)

 CPU time : 0.01

*
;; Include background knowledge about Tino.  Assume that his parent class is human and
;; that human is a basic level.
(describe (assert member (build lex "Tino")
                    class (build lex "human")))

(M13! (CLASS (M11 (LEX human))) (MEMBER (M7 (LEX Tino))))

(M13!)

 CPU time : 0.00

*
;; Model the fact that human is a basic level category.  This follows Ehrlich's case
;; frames
(describe (assert object1 (build lex "human")
                    rel ("ISA")
                    object2 (build lex "basic ctgy")))

(M14! (OBJECT1 (M11 (LEX human))) (OBJECT2 (M9 (LEX basic ctgy))) (REL ISA))

(M14!)
```

```
 CPU time : 0.00

*
;; Define throw as a transitive verb (at least in the context of the sentence)
(describe (assert object (build lex "throw")
                         property (build lex "bitransitive")))

(M16! (OBJECT (M5 (LEX throw))) (PROPERTY (M15 (LEX bitransitive))))

(M16!)

 CPU time : 0.00

*
;; The moment of truth...
^(
--> defn_verb 'throw)
((A (human) CAN VERB A (baseball) TO A (human) RESULT= NIL ENABLED BY= NIL)
 ((SUBJECT FOUND BY BASE_CAT_SUBJ) (OBJECT FOUND BY BASE_CAT_OBJ)
  (INDIRECT OBJECT FOUND BY BASE_CAT_INDOBJ) (NO CAUSES FOUND)
  (NO EFFECTS FOUND)))



 CPU time : 0.17

*
(resetnet)

Net reset - Relations and paths are still defined


 CPU time : 0.02

*
.....................................................................................
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;; Example Five - Demonstration of Ehrlich's verb algorithm with multiple instances
;; of the same verb.
;; The sentences are:
;; "Derek threw the baseball to Tino."
;; "Bobo throws the rubber ball."
.....................................................................................
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

;; Include background knowledge about a baseball.  Assume that ball is its parent class
;; and that ball is basic level.
(describe (assert subclass (build lex "baseball")
                         superclass (build lex "ball") ))

(M3! (SUBCLASS (M1 (LEX baseball))) (SUPERCLASS (M2 (LEX ball))))

(M3!)

 CPU time : 0.00

*
;; Make reference to the baseball without specifying which baseball it is.
```

```
(describe (assert member #baseball
                    class (build lex "baseball")))

(M4! (CLASS (M1 (LEX baseball))) (MEMBER B1))

(M4!)

 CPU time : 0.00

*
;; "Derek threw the baseball."
;; Represent the sentence without making reference to the fact that it is, at first
;; appearance, a transitive verb.  This shows how Ehrlich's algorithim treats verbs
;; not labeled as bitransitive, transitive, etc.

(describe (assert act (build lex "throw")
                    agent (build lex "Derek")
                    object *baseball
                    indobj(build lex "Tino")))

(M8! (ACT (M5 (LEX throw))) (AGENT (M6 (LEX Derek))) (INDOBJ (M7 (LEX Tino)))
 (OBJECT B1))

(M8!)

 CPU time : 0.01

*
;; Model the fact that ball is a basic level category.  This follows Ehrlich's case frames.
(describe (assert object1 (build lex "baseball")
                    rel ("ISA")
                    object2 (build lex "basic ctgy")))

(M10! (OBJECT1 (M1 (LEX baseball))) (OBJECT2 (M9 (LEX basic ctgy))) (REL ISA))

(M10!)

 CPU time : 0.00

*
;; Include background knowledge about Derek.  Assume that his parent class is human and
;; that human is a basic level.
(describe (assert member (build lex "Derek")
                    class (build lex "human")))

(M12! (CLASS (M11 (LEX human))) (MEMBER (M6 (LEX Derek))))

(M12!)

 CPU time : 0.01

*
;; Include background knowledge about Tino.  Assume that his parent class is human and
;; that human is a basic level.
(describe (assert member (build lex "Tino")
                    class (build lex "human")))
```

```
(M13! (CLASS (M11 (LEX human))) (MEMBER (M7 (LEX Tino))))

(M13!)

 CPU time : 0.00

*
;; Model the fact that human is a basic level category.  This follows Ehrlich's case
;; frames
(describe (assert object1 (build lex "human")
                          rel ("ISA")
                          object2 (build lex "basic ctgy")))

(M14! (OBJECT1 (M11 (LEX human))) (OBJECT2 (M9 (LEX basic ctgy))) (REL ISA))

(M14!)

 CPU time : 0.00

*
;; Define throw as a transitive verb (at least in the context of the sentence)
(describe (assert object (build lex "throw")
                          property (build lex "bitransitive")))

(M16! (OBJECT (M5 (LEX throw))) (PROPERTY (M15 (LEX bitransitive))))

(M16!)

 CPU time : 0.00

*
;; "Bobo throws the rubber ball"
;; Here assume that Bobo is a chimp.  Assume that chimp is a basic level category as well
;; as rubber ball.
;; Note that the sentence uses throw in a transitive sense.  Will the algoritm make a distinction
;; between the type of throwing a human can do and that which a chimp can do?

;; Include background knowledge about a rubber ball.  Assume that ball is its parent class.
(describe (assert subclass (build lex "rubber ball")
                          superclass (build lex "ball") ))

(M18! (SUBCLASS (M17 (LEX rubber ball))) (SUPERCLASS (M2 (LEX ball))))

(M18!)

 CPU time : 0.00

*
;; Make reference to the baseball without specifying which baseball it is.
(describe (assert member #rubberBall
                          class (build lex "rubber ball")))

(M19! (CLASS (M17 (LEX rubber ball))) (MEMBER B2))

(M19!)
```

```
 CPU time : 0.00

*
;; Model the action.
(describe (assert act (build lex "throw")
                      agent (build lex "Bobo")
                      object *rubberBall
                      ))

(M21! (ACT (M5 (LEX throw))) (AGENT (M20 (LEX Bobo))) (OBJECT B2))

(M21!)

 CPU time : 0.00

*
;; Model the fact that rubberball is a basic level category.
;; This follows Ehrlich's case frames.
(describe (assert object1 (build lex "rubber ball")
                      rel ("ISA")
                      object2 (build lex "basic ctgy")))

(M22! (OBJECT1 (M17 (LEX rubber ball))) (OBJECT2 (M9 (LEX basic ctgy)))
 (REL ISA))

(M22!)

 CPU time : 0.00

*
;; Model that Bobo is a chimp.
(describe (assert member (build lex "Bobo")
                      class (build lex "chimp")))

(M24! (CLASS (M23 (LEX chimp))) (MEMBER (M20 (LEX Bobo))))

(M24!)

 CPU time : 0.01

*
;; Model that chimp is a basic level category.
(describe (assert object1 (build lex "chimp")
                      rel ("ISA")
                      object2 (build lex "basic ctgy")))

(M25! (OBJECT1 (M23 (LEX chimp))) (OBJECT2 (M9 (LEX basic ctgy))) (REL ISA))

(M25!)

 CPU time : 0.00

*
;; The moment of truth...
^(
```

```
--> defn_verb 'throw)
((A (human chimp) CAN VERB A (baseball rubber ball) TO A (human) RESULT= NIL
  ENABLED BY= NIL)
 ((SUBJECT FOUND BY BASE_CAT_SUBJ) (OBJECT FOUND BY BASE_CAT_OBJ)
  (INDIRECT OBJECT FOUND BY BASE_CAT_INDOBJ) (NO CAUSES FOUND)
  (NO EFFECTS FOUND)))



 CPU time : 0.19

*
(resetnet)

Net reset - Relations and paths are still defined


 CPU time : 0.02

*
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;; Example Six - Demonstration of Ehrlich's verb algorithm with a verb used multiple
;; times in both transitive and bitransitive forms.  However, the transitive and
;; bitransitive lexes are set for the verb each time!  Will Ehrlich's algorithm
;; differentiate between the two?
;; The sentences are:
;; "Derek threw the baseball to Tino."
;; "Bobo throws the rubber ball."
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

;; Include background knowledge about a baseball.  Assume that ball is its parent class
;; and that ball is basic level.
(describe (assert subclass (build lex "baseball")
                  superclass (build lex "ball") ))

(M3! (SUBCLASS (M1 (LEX baseball))) (SUPERCLASS (M2 (LEX ball))))

(M3!)

 CPU time : 0.00

*
;; Make reference to the baseball without specifying which baseball it is.
(describe (assert member #baseball
                  class (build lex "baseball")))

(M4! (CLASS (M1 (LEX baseball))) (MEMBER B1))

(M4!)

 CPU time : 0.00

*
;; "Derek threw the baseball."
;; Represent the sentence without making reference to the fact that it is, at first
;; appearance, a transitive verb.  This shows how Ehrlich's algorithim treats verbs
```

```
;; not labeled as bitransitive, transitive, etc.

(describe (assert act (build lex "throw")
                      agent (build lex "Derek")
                      object *baseball
                      indobj(build lex "Tino")))

(M8! (ACT (M5 (LEX throw))) (AGENT (M6 (LEX Derek))) (INDOBJ (M7 (LEX Tino)))
 (OBJECT B1))

(M8!)

 CPU time : 0.01

*
;; Model the fact that ball is a basic level category.  This follows Ehrlich's case frames.
(describe (assert object1 (build lex "baseball")
                      rel ("ISA")
                      object2 (build lex "basic ctgy")))

(M10! (OBJECT1 (M1 (LEX baseball))) (OBJECT2 (M9 (LEX basic ctgy))) (REL ISA))

(M10!)

 CPU time : 0.00

*
;; Include background knowledge about Derek.  Assume that his parent class is human and
;; that human is a basic level.
(describe (assert member (build lex "Derek")
                      class (build lex "human")))

(M12! (CLASS (M11 (LEX human))) (MEMBER (M6 (LEX Derek))))

(M12!)

 CPU time : 0.01

*
;; Include background knowledge about Tino.  Assume that his parent class is human and
;; that human is a basic level.
(describe (assert member (build lex "Tino")
                      class (build lex "human")))

(M13! (CLASS (M11 (LEX human))) (MEMBER (M7 (LEX Tino))))

(M13!)

 CPU time : 0.00

*
;; Model the fact that human is a basic level category.  This follows Ehrlich's case
;; frames
(describe (assert object1 (build lex "human")
                      rel ("ISA")
                      object2 (build lex "basic ctgy")))
```

```
(M14! (OBJECT1 (M11 (LEX human))) (OBJECT2 (M9 (LEX basic ctgy))) (REL ISA))

(M14!)

 CPU time : 0.00

*
;; Define throw as a transitive verb (at least in the context of the sentence)
(describe (assert object (build lex "throw")
                         property (build lex "bitransitive")))

(M16! (OBJECT (M5 (LEX throw))) (PROPERTY (M15 (LEX bitransitive))))

(M16!)

 CPU time : 0.00

*
;; "Bobo throws the rubber ball"
;; Here assume that Bobo is a chimp.  Assume that chimp is a basic level category as well
;; as rubber ball.
;; Note that the sentence uses throw in a transitive sense.  Will the algoritm make a distinction
;; between the type of throwing a human can do and that which a chimp can do?

;; Include background knowledge about a rubber ball.  Assume that ball is its parent class.
(describe (assert subclass (build lex "rubber ball")
                         superclass (build lex "ball") ))

(M18! (SUBCLASS (M17 (LEX rubber ball))) (SUPERCLASS (M2 (LEX ball))))

(M18!)

 CPU time : 0.01

*
;; Make reference to the baseball without specifying which baseball it is.
(describe (assert member #rubberBall
                         class (build lex "rubber ball")))

(M19! (CLASS (M17 (LEX rubber ball))) (MEMBER B2))

(M19!)

 CPU time : 0.00

*
;; Model the action.
(describe (assert act (build lex "throw")
                         agent (build lex "Bobo")
                         object *rubberBall
                         ))

(M21! (ACT (M5 (LEX throw))) (AGENT (M20 (LEX Bobo))) (OBJECT B2))

(M21!)
```

```
 CPU time : 0.00

*
;; Define throw as a transitive verb.  Now it has been defined in two ways.  Will this affect
;; Ehrlich's algorithm?
(describe (assert object (build lex "throw")
                         property (build lex "transitive")))

(M23! (OBJECT (M5 (LEX throw))) (PROPERTY (M22 (LEX transitive))))

(M23!)

 CPU time : 0.01

*
;; Model the fact that rubberball is a basic level category.
;; This follows Ehrlich's case frames.
(describe (assert object1 (build lex "rubber ball")
                         rel ("ISA")
                         object2 (build lex "basic ctgy")))

(M24! (OBJECT1 (M17 (LEX rubber ball))) (OBJECT2 (M9 (LEX basic ctgy)))
 (REL ISA))

(M24!)

 CPU time : 0.00

*
;; Model that Bobo is a chimp.
(describe (assert member (build lex "Bobo")
                         class (build lex "chimp")))

(M26! (CLASS (M25 (LEX chimp))) (MEMBER (M20 (LEX Bobo))))

(M26!)

 CPU time : 0.00

*
;; Model that chimp is a basic level category.
(describe (assert object1 (build lex "chimp")
                         rel ("ISA")
                         object2 (build lex "basic ctgy")))

(M27! (OBJECT1 (M25 (LEX chimp))) (OBJECT2 (M9 (LEX basic ctgy))) (REL ISA))

(M27!)

 CPU time : 0.00

*
;; The moment of truth...
^(
--> defn_verb 'throw)
```

```
((A (human chimp) CAN VERB A (baseball rubber ball) TO A (human) RESULT= NIL
  ENABLED BY= NIL)
 ((SUBJECT FOUND BY BASE_CAT_SUBJ) (OBJECT FOUND BY BASE_CAT_OBJ)
  (INDIRECT OBJECT FOUND BY BASE_CAT_INDOBJ) (NO CAUSES FOUND)
  (NO EFFECTS FOUND)))



 CPU time : 0.19

*
(resetnet)

Net reset - Relations and paths are still defined


 CPU time : 0.01

*
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;; Example Seven - Demonstrate Ehrlich's algorithm with a verb that has a consequence
;; defined for it.
;; The sentences are:
;; "Derek threw the baseball to Tino."
;; "Bobo throws the rubber ball."
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

;; Include background knowledge about a baseball.  Assume that ball is its parent class
;; and that ball is basic level.
(describe (assert subclass (build lex "baseball")
                  superclass (build lex "ball") ))

(M3! (SUBCLASS (M1 (LEX baseball))) (SUPERCLASS (M2 (LEX ball))))

(M3!)

 CPU time : 0.00

*
;; Make reference to the baseball without specifying which baseball it is.
(describe (assert member #baseball
                  class (build lex "baseball")))

(M4! (CLASS (M1 (LEX baseball))) (MEMBER B1))

(M4!)

 CPU time : 0.00

*
;; "Derek threw the baseball."
;; Represent the sentence without making reference to the fact that it is, at first
;; appearance, a transitive verb.  This shows how Ehrlich's algorithim treats verbs
;; not labeled as bitransitive, transitive, etc.

(describe (assert act (build lex "throw")
```

```
                    agent (build lex "Derek")
                    object *baseball
                    indobj(build lex "Tino")))

(M8! (ACT (M5 (LEX throw))) (AGENT (M6 (LEX Derek))) (INDOBJ (M7 (LEX Tino)))
 (OBJECT B1))

(M8!)

 CPU time : 0.01

*
;; Model the fact that ball is a basic level category.  This follows Ehrlich's case frames.
(describe (assert object1 (build lex "baseball")
                    rel ("ISA")
                    object2 (build lex "basic ctgy")))

(M10! (OBJECT1 (M1 (LEX baseball))) (OBJECT2 (M9 (LEX basic ctgy))) (REL ISA))

(M10!)

 CPU time : 0.00

*
;; Include background knowledge about Derek.  Assume that his parent class is human and
;; that human is a basic level.
(describe (assert member (build lex "Derek")
                    class (build lex "human")))

(M12! (CLASS (M11 (LEX human))) (MEMBER (M6 (LEX Derek))))

(M12!)

 CPU time : 0.00

*
;; Include background knowledge about Tino.  Assume that his parent class is human and
;; that human is a basic level.
(describe (assert member (build lex "Tino")
                    class (build lex "human")))

(M13! (CLASS (M11 (LEX human))) (MEMBER (M7 (LEX Tino))))

(M13!)

 CPU time : 0.01

*
;; Model the fact that human is a basic level category.  This follows Ehrlich's case
;; frames
(describe (assert object1 (build lex "human")
                    rel ("ISA")
                    object2 (build lex "basic ctgy")))

(M14! (OBJECT1 (M11 (LEX human))) (OBJECT2 (M9 (LEX basic ctgy))) (REL ISA))
```

```
(M14!)

 CPU time : 0.00

*
;; Define throw as a transitive verb (at least in the context of the sentence)
(describe (assert object (build lex "throw")
                         property (build lex "bitransitive")))

(M16! (OBJECT (M5 (LEX throw))) (PROPERTY (M15 (LEX bitransitive))))

(M16!)

 CPU time : 0.00

*
;; "Bobo throws the rubber ball"
;; Here assume that Bobo is a chimp.  Assume that chimp is a basic level category as well
;; as rubber ball.
;; Note that the sentence uses throw in a transitive sense.  Will the algoritm make a distinction
;; between the type of throwing a human can do and that which a chimp can do?

;; Include background knowledge about a rubber ball.  Assume that ball is its parent class.
(describe (assert subclass (build lex "rubber ball")
                         superclass (build lex "ball") ))

(M18! (SUBCLASS (M17 (LEX rubber ball))) (SUPERCLASS (M2 (LEX ball))))

(M18!)

 CPU time : 0.00

*
;; Make reference to the baseball without specifying which baseball it is.
(describe (assert member #rubberBall
                         class (build lex "rubber ball")))

(M19! (CLASS (M17 (LEX rubber ball))) (MEMBER B2))

(M19!)

 CPU time : 0.00

*
;; Model the action.
(describe (assert act (build lex "throw")
                         agent (build lex "Bobo")
                         object *rubberBall
                         ))

(M21! (ACT (M5 (LEX throw))) (AGENT (M20 (LEX Bobo))) (OBJECT B2))

(M21!)

 CPU time : 0.01
```

```
*
;; Define throw as a transitive verb.  Now it has been defined in two ways.  Will this affect
;; Ehrlich's algorithm?
(describe (assert object (build lex "throw")
                         property (build lex "transitive")))

(M23! (OBJECT (M5 (LEX throw))) (PROPERTY (M22 (LEX transitive))))

(M23!)

 CPU time : 0.00

*
;; Model the fact that rubberball is a basic level category.
;; This follows Ehrlich's case frames.
(describe (assert object1 (build lex "rubber ball")
                         rel ("ISA")
                         object2 (build lex "basic ctgy")))

(M24! (OBJECT1 (M17 (LEX rubber ball))) (OBJECT2 (M9 (LEX basic ctgy)))
 (REL ISA))

(M24!)

 CPU time : 0.00

*
;; Model that Bobo is a chimp.
(describe (assert member (build lex "Bobo")
                         class (build lex "chimp")))

(M26! (CLASS (M25 (LEX chimp))) (MEMBER (M20 (LEX Bobo))))

(M26!)

 CPU time : 0.00

*
;; Model that chimp is a basic level category.
(describe (assert object1 (build lex "chimp")
                         rel ("ISA")
                         object2 (build lex "basic ctgy")))

(M27! (OBJECT1 (M25 (LEX chimp))) (OBJECT2 (M9 (LEX basic ctgy))) (REL ISA))

(M27!)

 CPU time : 0.00

*
;; Create a path-based inference we will need for subclasses
;; Notice the results this has on the verb algorithms output!
(define-path class
          (compose class
                   (kstar (compose subclass- ! superclass))))
CLASS implied by the path (COMPOSE CLASS
```

```
                    (KSTAR (COMPOSE SUBCLASS- ! SUPERCLASS)))
CLASS- implied by the path (COMPOSE (KSTAR (COMPOSE SUPERCLASS- ! SUBCLASS))
                    CLASS-)


 CPU time : 0.00

*
;; Create a consequence (in Ehrlichs terms) for throwing.  This consequence is what
;; Ehrlich would have considered a life-rule2
;; "If a person x throws a ball then he lofts it in the air

(describe (assert forall ($h $b)
                    &ant((build member *h
                              class (build lex "human"))
                       (build member *b
                              class (build lex "ball"))
                       (build agent *h
                              act (build lex "throw")
                              object *b))
                    cq (build agent *h
                              act (build lex "loft")
                              object *b
                              location (build lex "through the air")
                              )
                    kn_cat ("life-rule.2")))

(M30! (FORALL V2 V1)
 (&ANT (P3 (ACT (M5 (LEX throw))) (AGENT V1) (OBJECT V2))
  (P2 (CLASS (M2 (LEX ball))) (MEMBER V2))
  (P1 (CLASS (M11 (LEX human))) (MEMBER V1)))
 (CQ
  (P4 (ACT (M28 (LEX loft))) (AGENT V1) (LOCATION (M29 (LEX through the air)))
   (OBJECT V2)))
 (KN_CAT life-rule.2))

(M30!)

 CPU time : 0.01

*
;; Proof that the rule is in working order
(describe (deduce agent $whoever act (build lex "loft")))

(M35! (ACT (M28 (LEX loft))) (AGENT (M6 (LEX Derek))))

(M35!)

 CPU time : 0.02

*

^(
--> defn_verb 'throw)
((A (human chimp) CAN VERB A (baseball ball rubber ball) TO A (human) RESULT=
  (P4) ENABLED BY= NIL)
```

((SUBJECT FOUND BY BASE_CAT_SUBJ) (OBJECT FOUND BY BASE_CAT_OBJ)
(INDIRECT OBJECT FOUND BY BASE_CAT_INDOBJ) (CAUSE FOUND CQS FROM &ANTS)
(NO EFFECTS FOUND)))

CPU time : 0.24

*
(resetnet)

Net reset - Relations and paths are still defined

CPU time : 0.02

*

End of /home/csgrad/sood/cse663/demos/throw_well.demo demonstration.

CPU time : 1.75

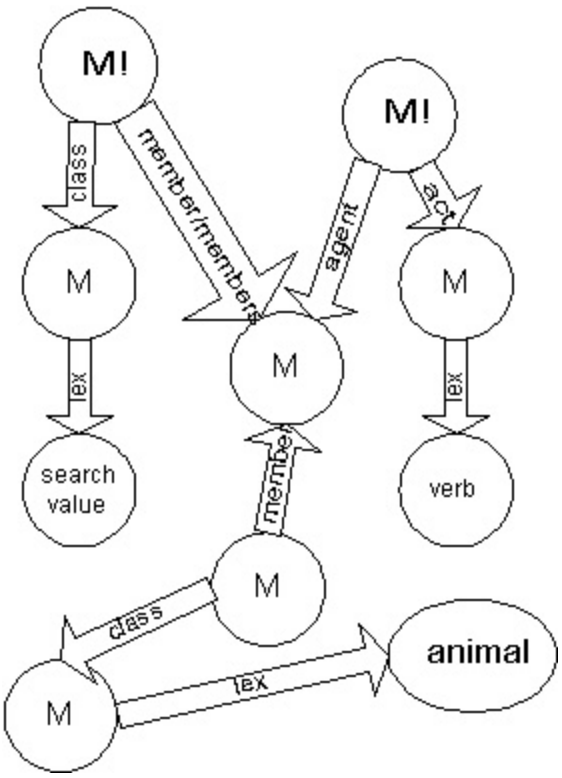
*

8.4 Search Paths for verb algorithm

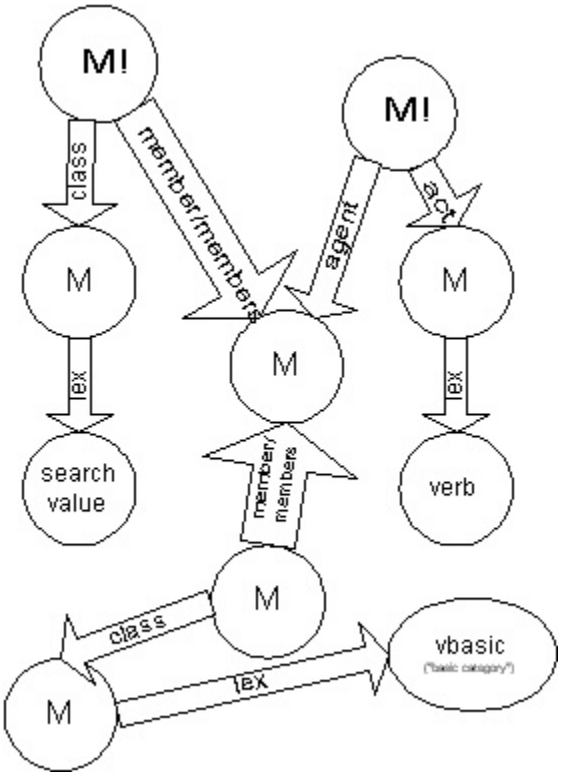
The following search paths are those that my version of the algorithm searches for. Note that these are the same as those that Karen Ehrlich searches for with the exception of the following cases:

1. In the functions some_cat_obj and some_cat_indobj, she writes the same statement twice. I think that the second one should read objects1 and "ARE".
2. In some_cat_obj, she uses agent-, instead of object-.

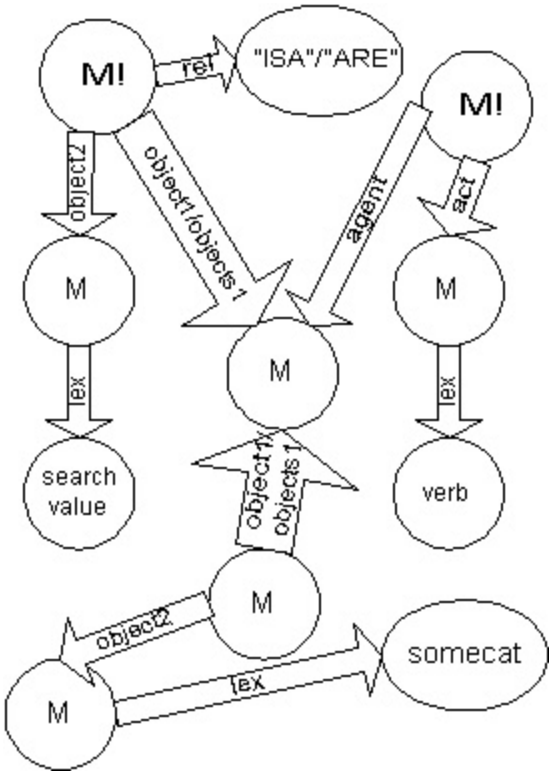
anim_subj



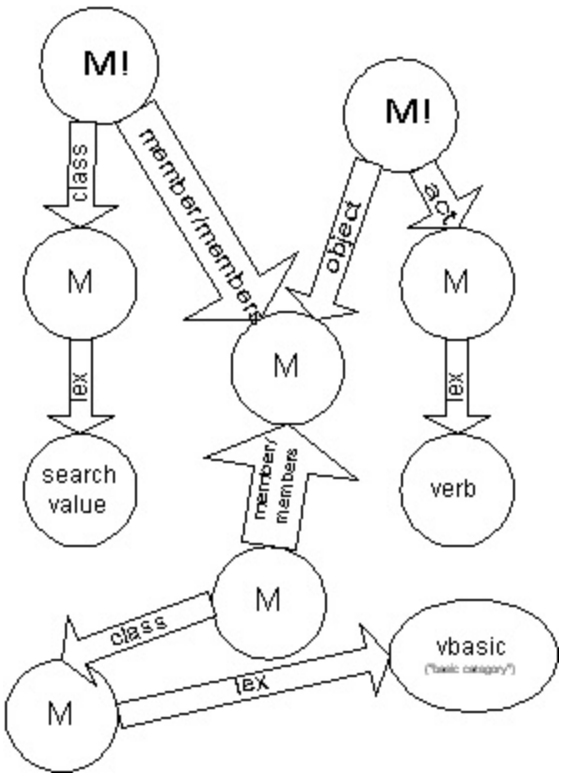
base_cat_subj



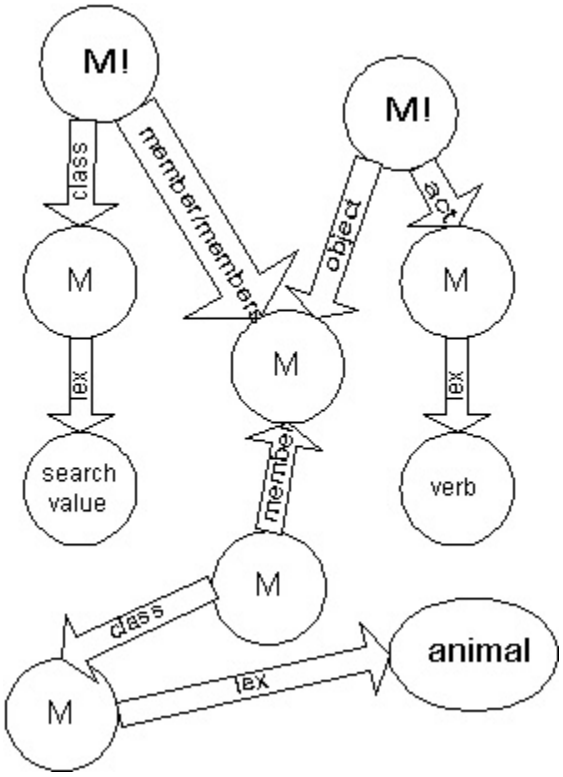
some_cat_subj



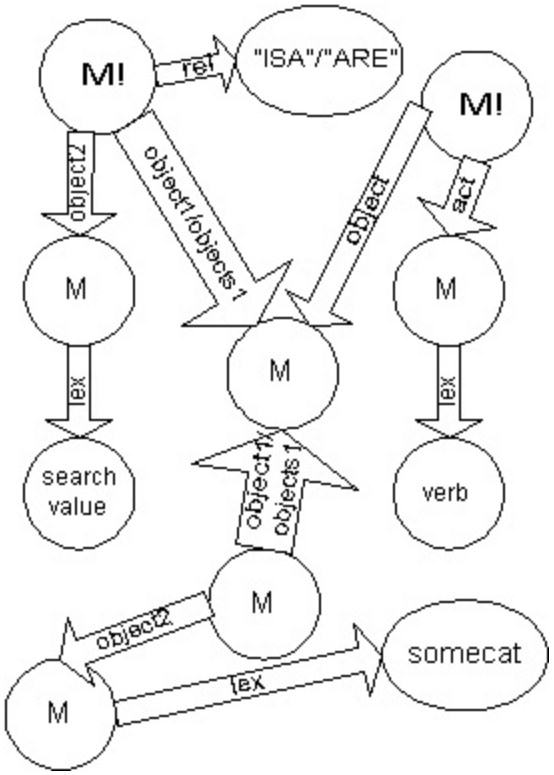
base_cat_obj


M!
M!
class
member/members
object
act
M
M
M
lex
lex
search value
verb
member/members
M
class
vbasic
("basic category")
lex
M

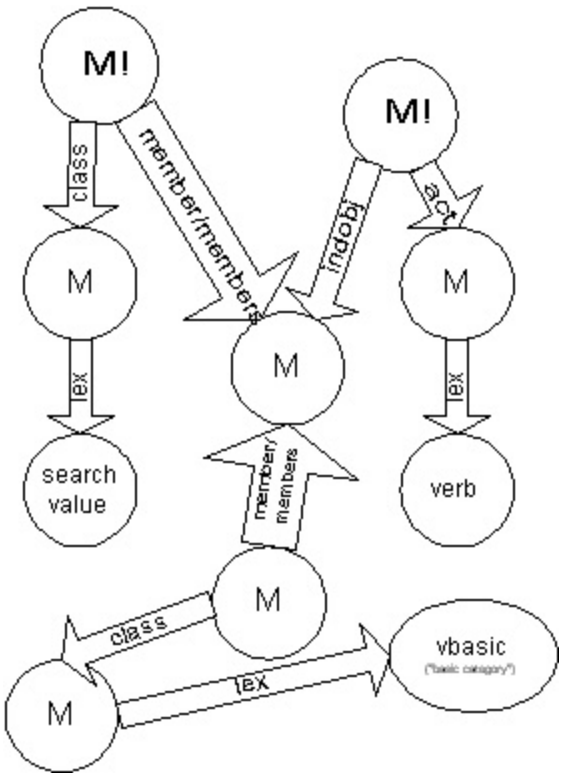
anim_obj


M!
M!
class
member/members
object
act
M
M
M
lex
lex
search value
verb
member
M
class
animal
lex
M

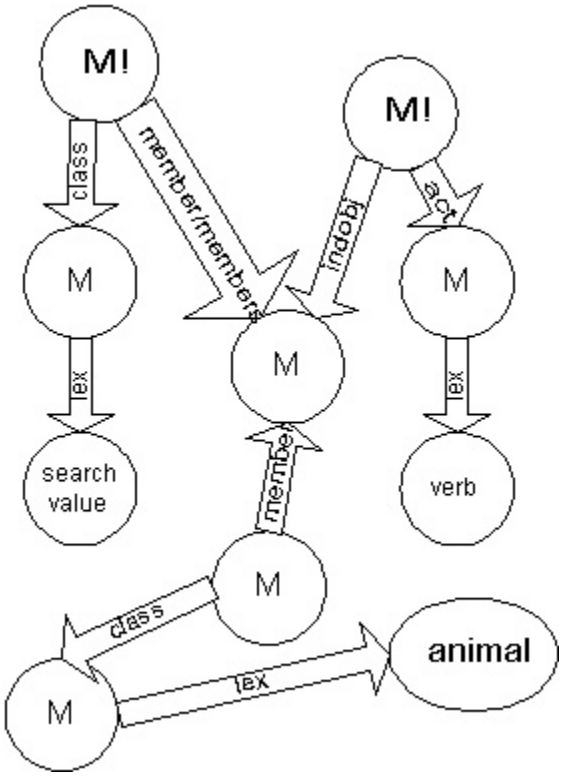
some_cat_obj



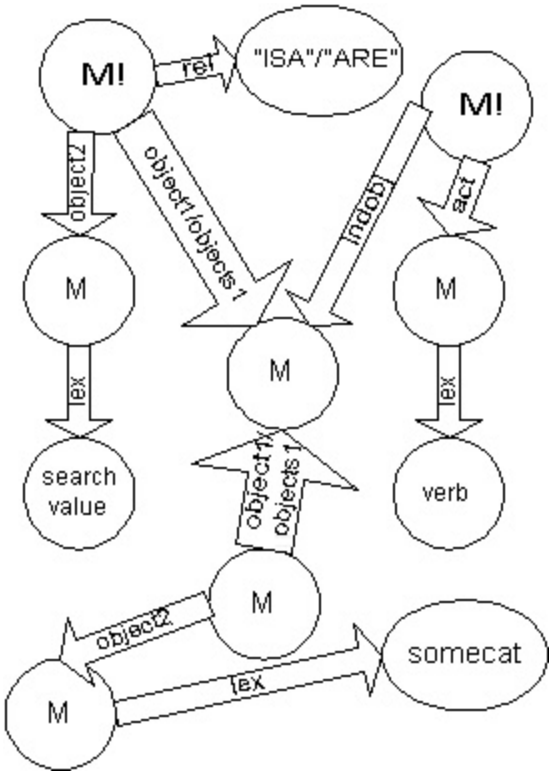
base_cat_indobj


M!
M!
class
member/members
indobj
act
M
M
M
lex
lex
search value
verb
member/members
M
class
vbasic
("basic category")
lex
M

anim_indobj

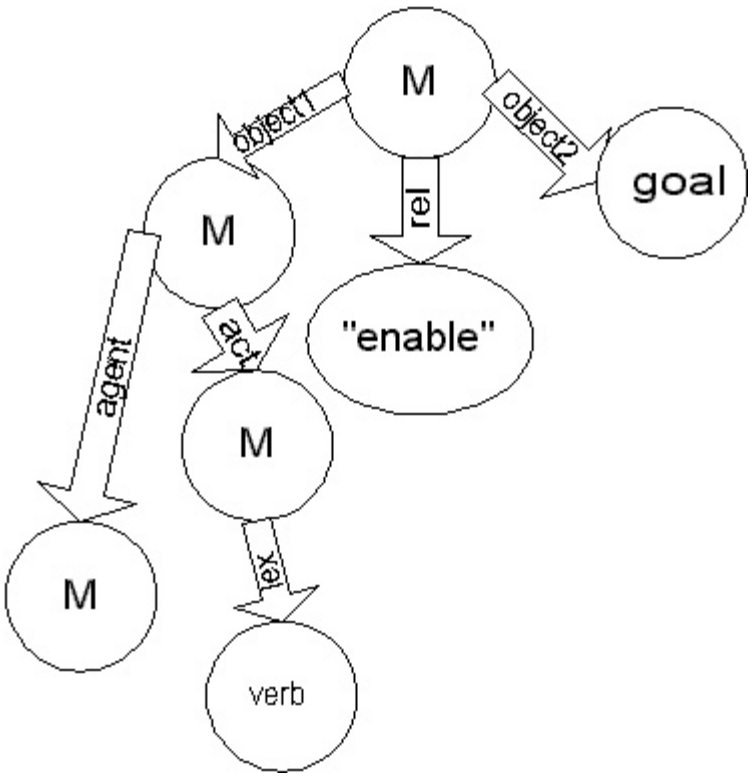

M!
M!
class
member/members
indobj
act
M
M
M
lex
lex
search value
verb
member
M
class
animal
lex
M

some_cat_indobj

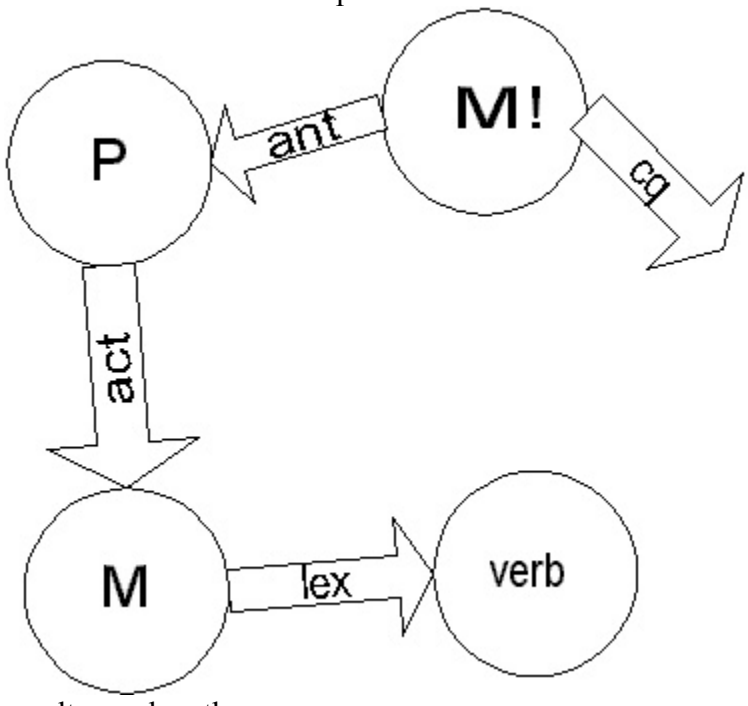


cause

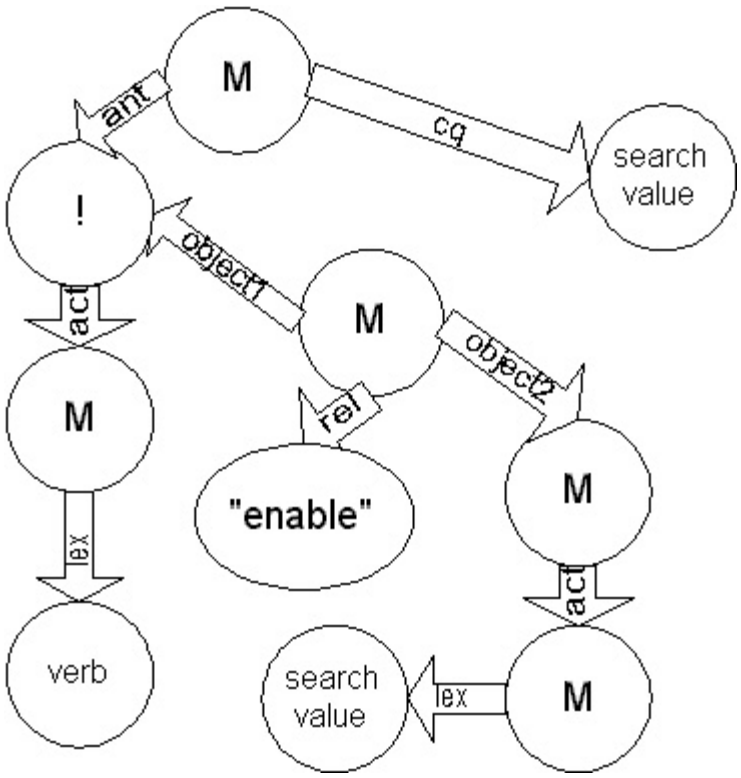
- condition 1 search path



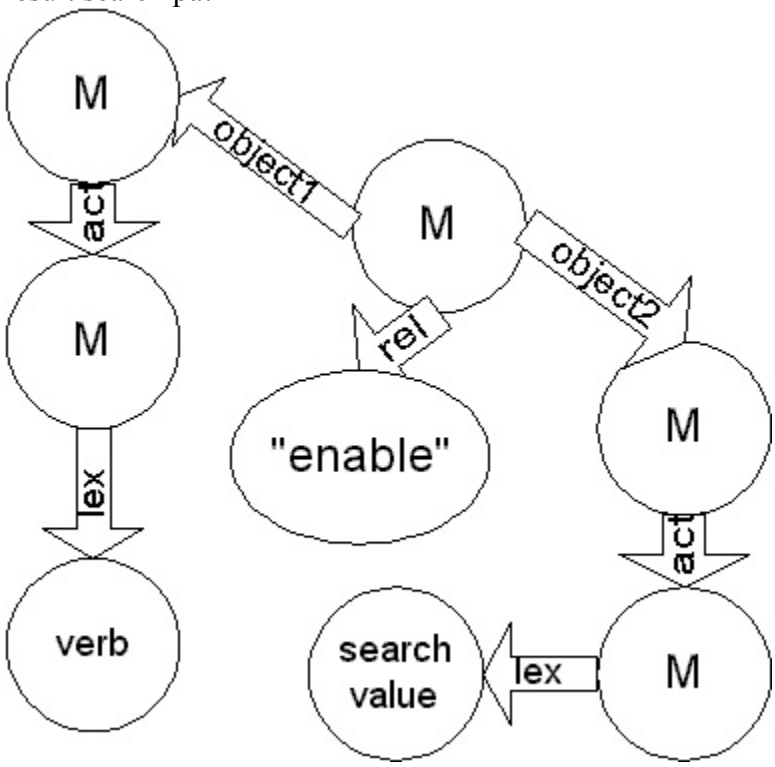
- nested condition 1 search path



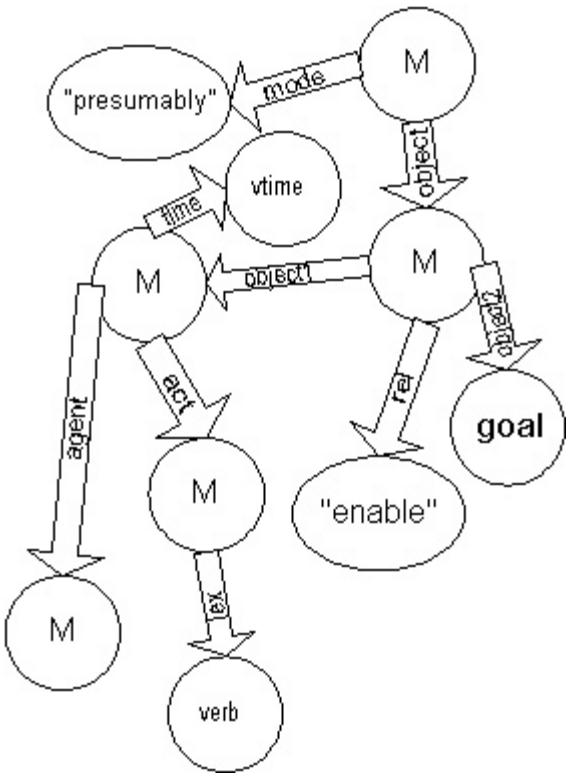
- result search path



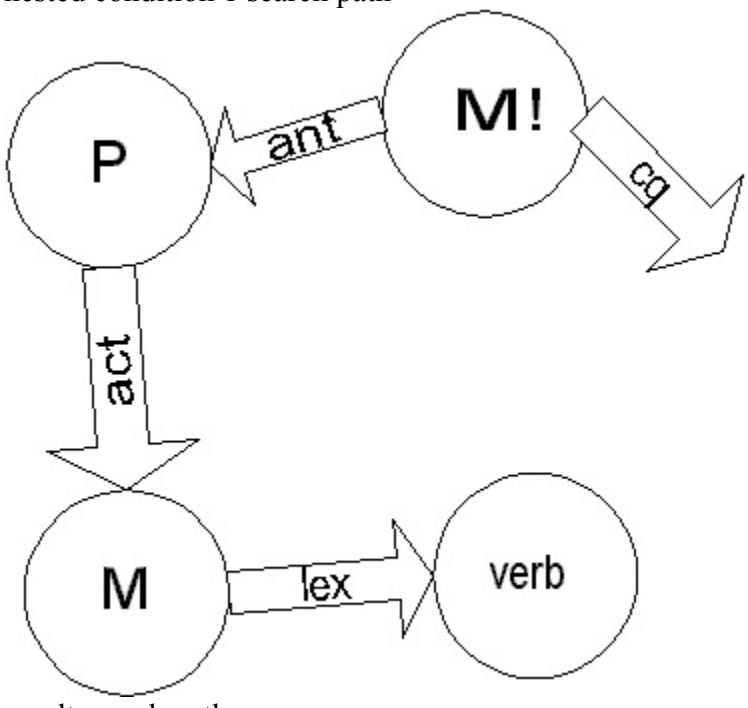
- else
  result search path



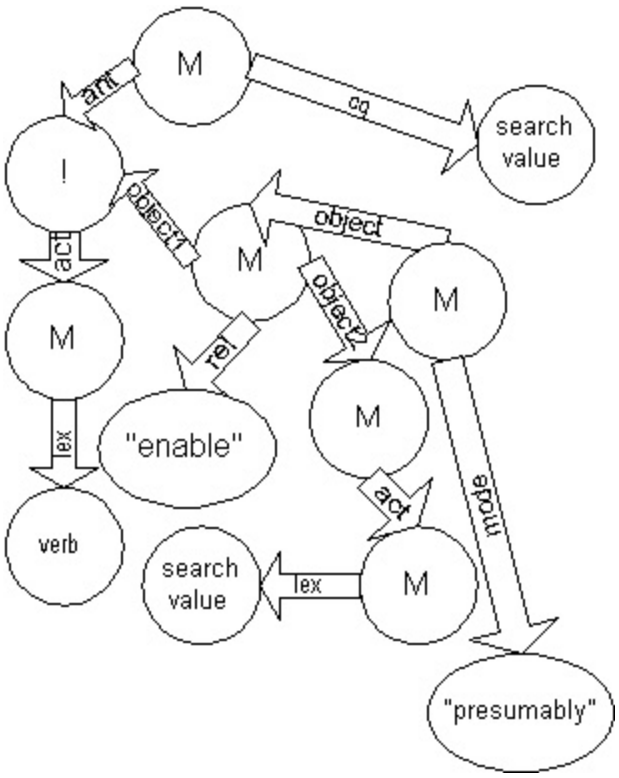
- else
- condition 2 search path



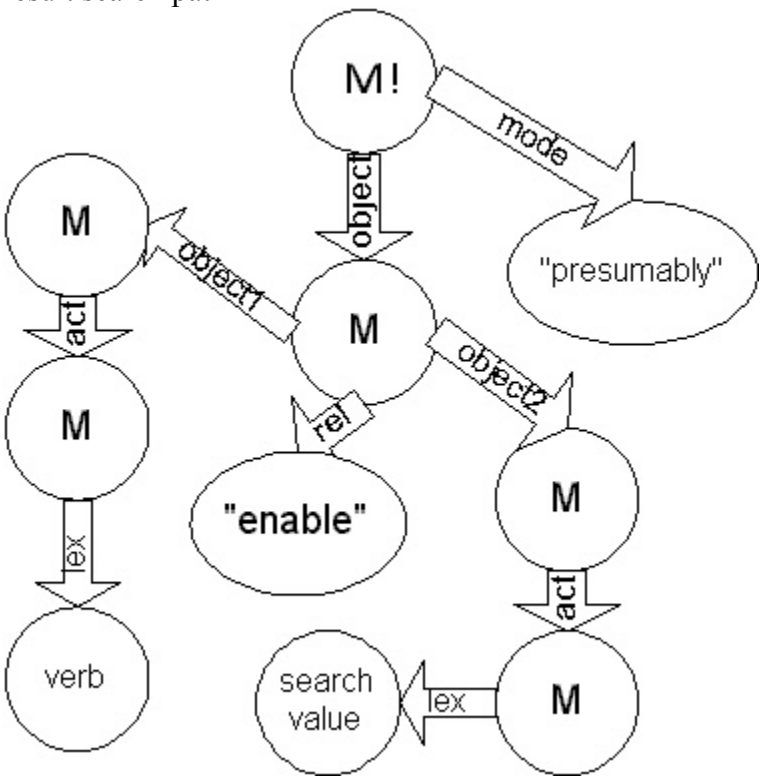
- nested condition 1 search path



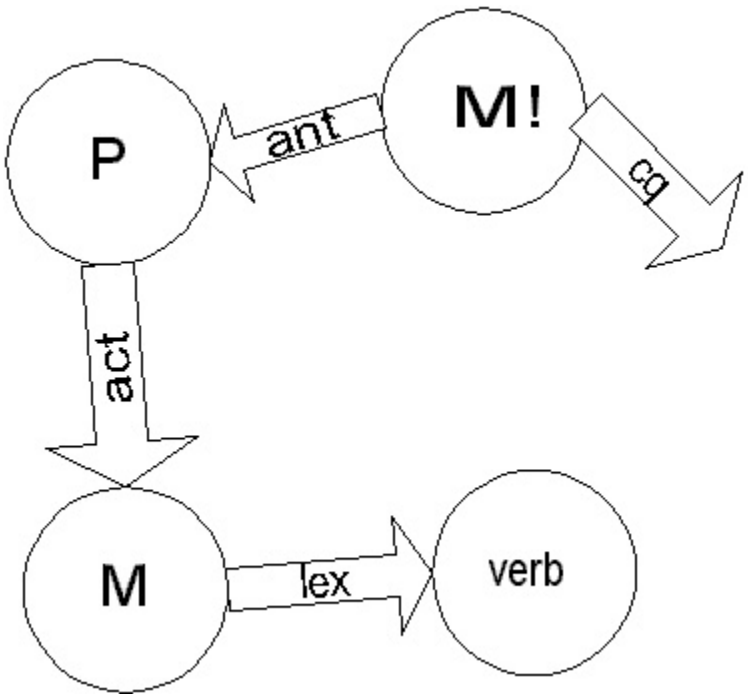
- result search path

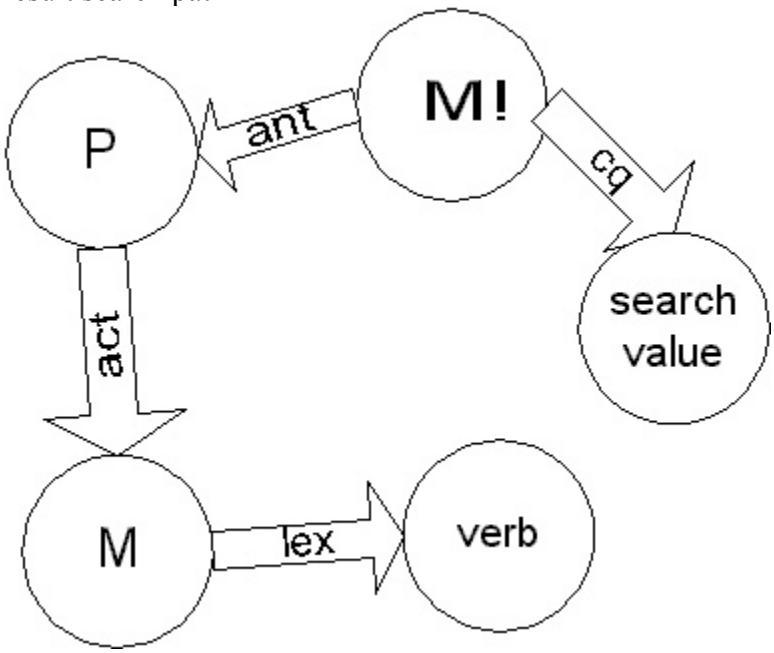


- else
  result search path

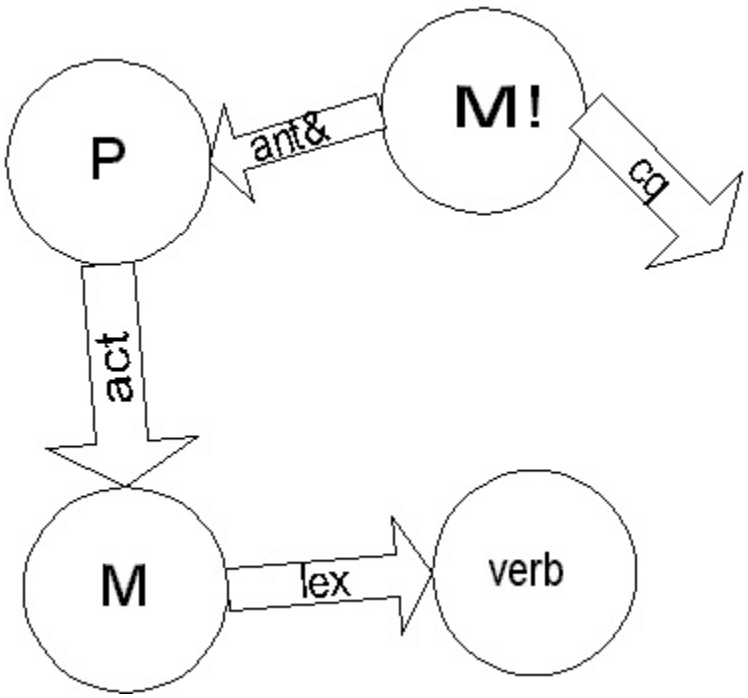


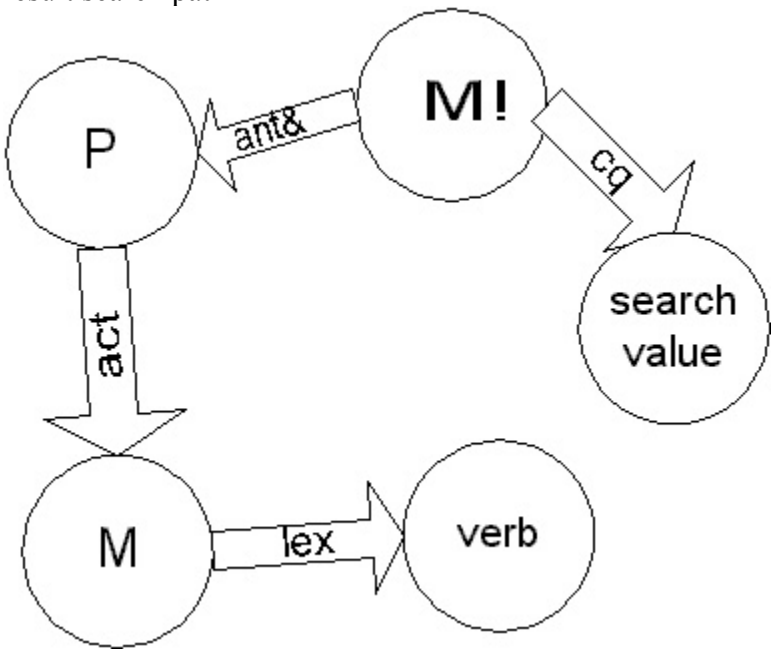
- else
- condition 3 search path

- 

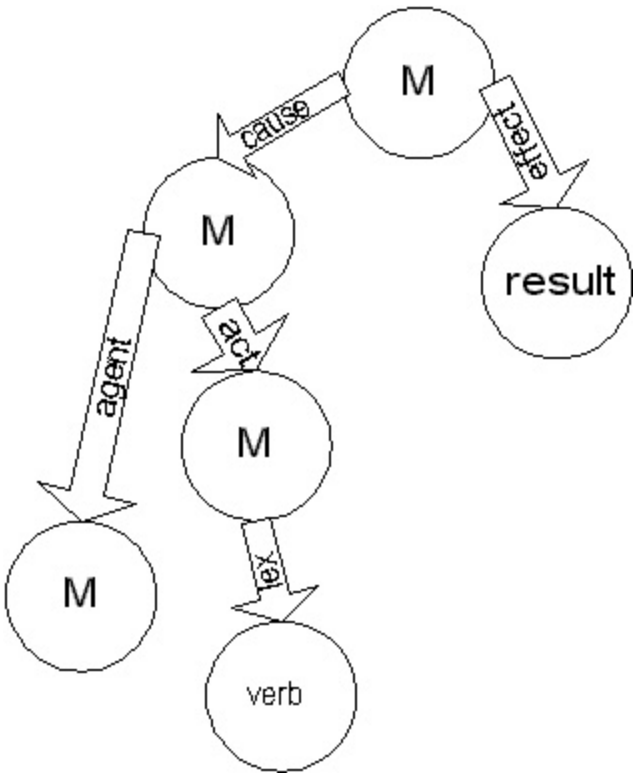
result search path
- 

- else
- condition 4 search path

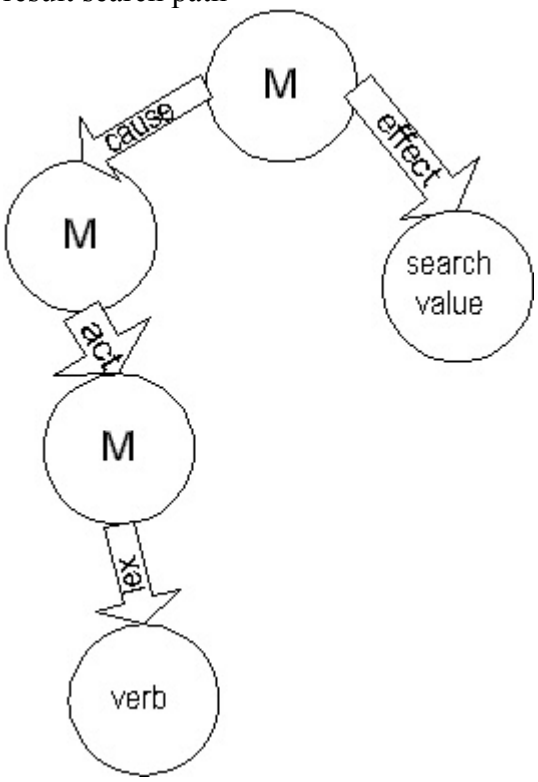
- 

result search path
- 

- else
- condition 5 search path

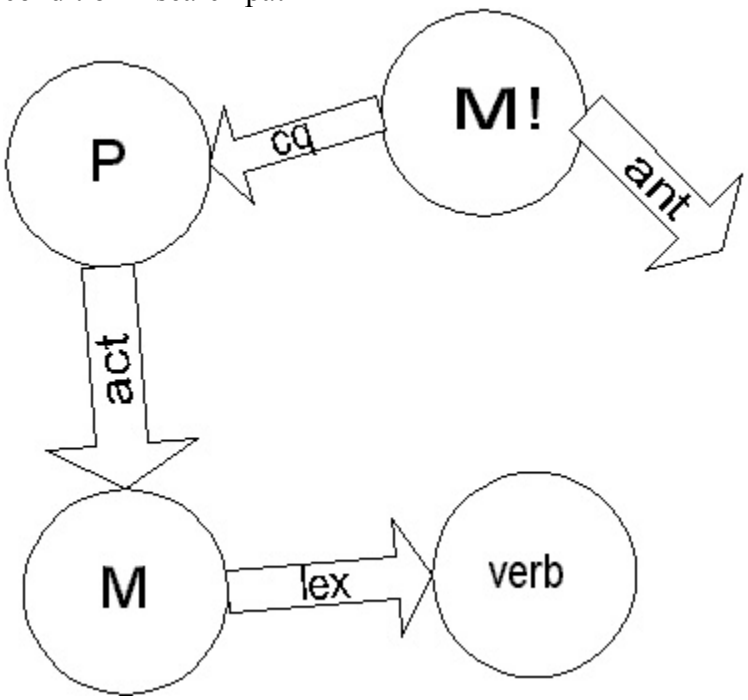


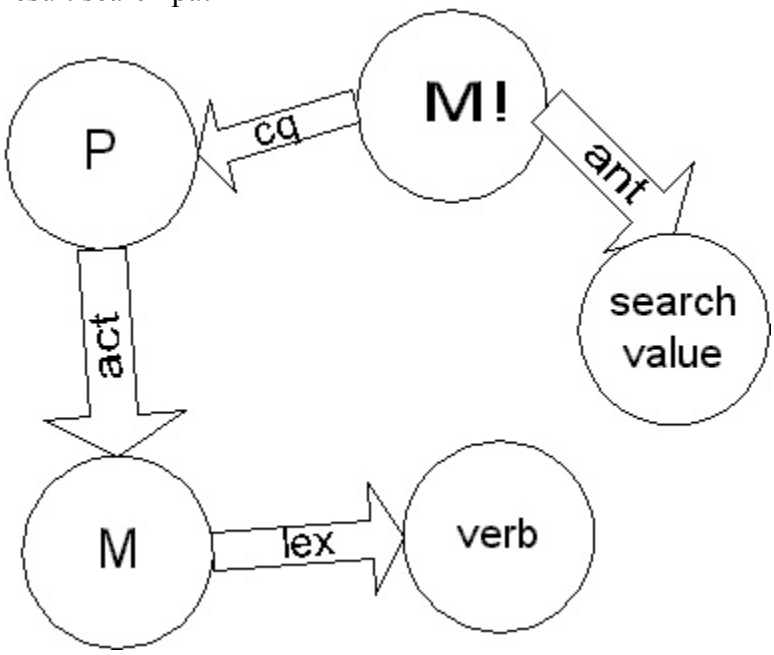
- result search path



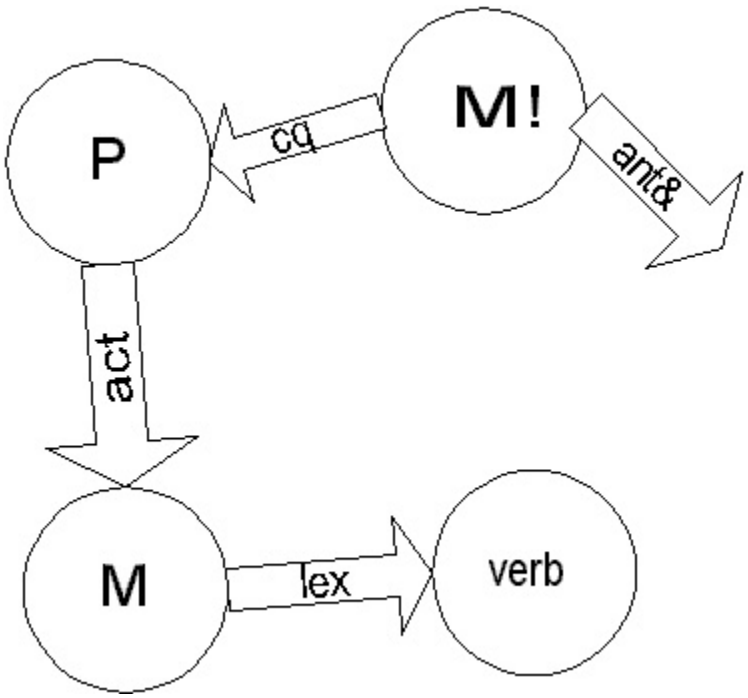
-

effect

- condition 1 search path
- 

result search path
- 

- else
- condition 2 search path

- 

result search path
-