A Computational Definition of “proliferate” Based on Context

Justin M. Del Vecchio

6 May 2002

CSE 663 - Advanced Knowledge Representation

Abstract

When people learn the meaning of a word from exposure to it in context they engage in contextual vocabulary acquisition. There are different ways people can guess definitions from context including incidental learning and using an explicit set of instructions to derive the meaning. This report analyzes a computational system, “Cassie”, that embraces the latter methodology. To demonstrate how she goes about contextual vocabulary acquisition Cassie is given a passage with a single unknown word, “proliferate”, and an accompanying set of appropriate background knowledge. This passage and background knowledge, encoded in Semantic Network Processing System User Language code, serve as Cassie’s mind. In order for Cassie to guess a verb’s meaning based on context she must be prompted to do so. This is the purpose of Ehrlich’s verb algorithm as it allows Cassie to apply a set of analytical steps in order to guess the meaning of a verb based on its context and her background knowledge. A thorough analysis of the passage and background design decisions and the algorithm’s output is provided to illustrate the steps the algorithm follows when generating a definition. A general demo for the verb algorithm is also included so important aspects of its functioning not touched on in the “proliferate” example are covered. The report concludes with an overall review of the algorithm

and a list of possible changes and enhancements that may improve it.

1 Introduction

Consider a passage that contains a word unknown to the reader. When the reader is confronted with the word she might use the context it is presented in and her personal background knowledge to create a definition for the unknown word. In doing so the reader engages in contextual vocabulary acquisition (CVA) to generate a definition for the word that may not be an exact dictionary definition and is open to further refinement when future instances of the word are encountered. Sternberg (1987) showed that readers who were given explicit instructions to figure out the meaning of words from context did better at defining new words than those who used no process at all or relied on word memorization. An analytical process for defining words based on context has been modeled on a computational level using the Cognitive Agent of the SNePS System-an Intelligent Entity (Cassie) and the noun/verb algorithm created by Ehrlich (1995). The model calls for Cassie to mimic the mind of the reader. She has Semantic Network Processing System (SNePS) representations of the passage as well as background information that the reader would be expected to have and use to arrive at deductions. Ehrlich's noun/verb algorithm, a combination of LISP and the SNePS User Language (SNePSUL), is run on the SNePS representations and yields a definition for the unknown word that is accurate based on the input information.

This report represents a text passage in SNePSUL where one unknown word exists: "proliferate". It also creates a representation for background knowledge pertinent to the passage. The representations are input into the noun/verb algorithm and the results analyzed. A demo of the verb algorithm using the verb "throw" is provided to demonstrate its functionality in a variety of situations, not simply the one instance for "proliferate". The generic "throw" demonstration shows the overall functionality of the verb algorithm while the "proliferate" case study shows the lengths to which a representation needs to be taken to maximize the algorithm's returned information.

2 CVA for Humans

When a person encounters an unknown word in text she has four (possibly more) choices for learning the word's meaning each of which has positives and negatives (Nagy, Herman & Anderson, 1985):

1. Ignore the word and continue reading the passage.
2. Being told the meaning of the word directly from another person, a dictionary, etc.
3. Incidentally learning the meaning of the word from its context.
4. Deriving the meaning of the word by following a set of explicit instructions that focus on analyzing the context.

The first option has the benefit of allowing the reader to continue reading and not bothering to stop and either guess a or research a definition. As Rapaport and Kibby (2000: 3) point out this method has severe limitations for science, mathematics, engineering or technology (SMET) texts where not understanding a current word decreases the likelihood of the person understanding the subsequent text.

The second option certainly provides for an exact definition but has a couple of obvious deficiencies. First, it takes time to stop and research a definition, time an average reader may not be willing to spend. Second, and more importantly, dictionary definitions do not cover all contextual uses. A reader may spend time futilely trying to couple the word's dictionary definition with its current contextual usage.

The third option stresses that with each encounter of the word in text the reader will be able to partially increase her knowledge of the word (Nagy, Herman & Anderson, 1985). When exposed to enough examples of the word a reasonably thorough definition of it will materialize. This option differs from option four in that here the reader is not intensely focusing on the word and applying a set of algorithmic steps to determine meaning. The reader obtains what information she can quickly and moves on hoping that future instances will create a more refined definition.

The fourth option proposes the reader review the word's context by using a set of explicit steps or instructions directing her to look for clues that will shed light on the word's meaning. This method was employed by Sternberg (1987) with very good results. Ehrlich's verb algorithm is similar to option four as it is essentially a set of steps that focuses on parts of the passage and background representations searching for clues in order to create a definition of a word from context.

Option four is primary focus of this report with Cassie mimicking the mind of a human. Her mind contains two related entities: First, the text passage that it is reading and second, a "toy" knowledge base that is a series of antecedent knowledge related to the text being read (Rapaport & Kibby, 2000: 7). Both the knowledge base and passage text are currently hand encoded and creating an automated parser for this information is a future goal of the research.

With a knowledge base and text representation Cassie contains the necessary information to hypothesize the definition of a word in a fashion similar to a human using a set of predefined steps. The verb algorithm is the query mechanism that essentially asks Cassie "What is the definition of the verb x based on everything you know?". The verb algorithm follows a set of analytical steps where it requests Cassie first report on the casual or enablement information and then examine the predicate structure. The generated definition is not incorporated into Cassie's background knowledge and is dynamic, changing each time Cassie is updated with additional pertinent text passages or background knowledge.

3 Overview of Ehrlich's Verb Algorithm

Ehrlich's verb algorithm is surprising simple with respect to the sequence of steps it uses to create a definition for a single input verb. The algorithm is interested in two main aspects of the verb: First, the verb's predicate structure and second, categorization of its arguments such as casual and enablement information (Ehrlich 1995: 96).

3.1 Predicate Structure

The complete predicate consists of the verb along with it complements and modifiers. So for the sentence "He runs along the riverbank", *runs along the riverbank* is the complete predicate and *he* is the subject. Ehrlich's verb algorithm decomposes this grammatical structure into its components (subject, action, object and indirect object) and then reports information about them. It uses a categorization system in order to format the results returned to the user. Categorization of the verb type can fall into one of four categories:

- **Bitransitive** - If an agent acts on an object and there is an indirect object of the action.

  Example: "John drove the car home."

- **Transitive** - If an agent acts on an object.

  Example: "John drove the car."

- **Reflexive** - If an agent acts on itself (after reflexive pronouns like myself, yourself, himself, etc.)

  Example: "John hated himself."

- **Intransitive** - If an agent acts.

  Example: "John fished in the stream."

It should be noted that it is up to the knowledge engineer to determine the type of each verb when creating SNePS representations. He must determine the verb type based on usage and assert this type using an object/property relation (see section 9.2 for reasons this may want to be changed). The representations in this report assign a verb type to each verb represented for completeness.

The algorithm is input a single verb and then searches for verb type in the following order: bitransitive, transitive, reflexive and intransitive. By "searching" the verb algorithm is looking for an act arc with the specific verb in question and a **OBJ ! PROPERTY** path emanating from it leading to one of the four verb categories. If a single instance of the verb is found to be bitransitive then all instances of the verb are evaluated and reported as bitransitive. If no bitransitive instances are found, transitive instances are searched and if one is found all instances are reported as being transitive. This continues until the intransitive case is reached. At this point no searching is done and all instances are reported as being intransitive.

After the verb algorithm finds the verb in question and evaluates its type it next looks at the ends of the agent, object and indobj arcs (dependent on type) to determine what information to return to the user. The arcs can be viewed as a definitional framework for the target verb with each individual arc a slot requiring information from each specific instance of the verb to fill it (Rapaport and Ehrlich 2000: 7).

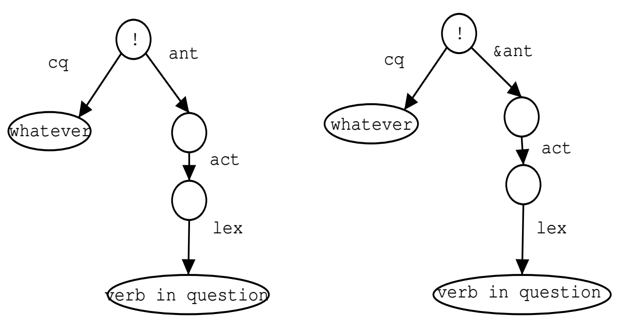
Labeling an instance of the verb as being bitransitive, transitive, reflexive or intransitive has major implications on how the results returned are formatted. The verb algorithm reports results based on the first type that it finds so if a single instance of the specific verb is labeled as bitransitive while 200 others are labeled as transitive the verb algorithm will report all results as if they were bitransitive. It makes no attempt to delineate between the different types and possibly separate them. The demonstration in the preceeding pages shows precisely how Ehrlich's verb algorithm goes about reporting.

3.2 Consequences and Effects

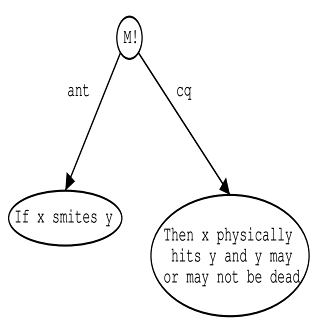
The second major component of the verb algorithm's output are casual and enablement information. Casual information refers to the consequences (effects) of an action taking place. Enablement information are conditions that exist after an action takes place that did not exist prior to it and were made possible by the action taking place.

In order to determine the consequences of actions, the algorithm searches ant or &ant arcs for instances of the verb in question and then backtracks to find the accompanying cq arcs. Whatever is at the end of the cq arc is considered a consequence of the action taking place.

The algorithm would search on the paths:



It wants to find a general relation where "If this verb happens these are the consequences." A simple example (not following pure SNePS coding practices):



Cause and effect is similar to the action/effect case frame. Ehrlich searches on the following path,

(Effect- Cause Act lex)

where Act is the verb in question and Effect is a result of the verb. The effect of the verb need not be a rule and can represent virtually anything. For instance if the sentence is “John ran out of the house and to the school” than an effect of “John running” is that “John is at school”. Where ant/cq are more geared toward a set of logical consequences or a brief story of describing what the action is all about, cause/effect is more subjective and open to anything that might be an effect of the verb.

3.3 A Demonstration of the Verb Algorithm Using “throw”

Here sentences will be modeled using SNePS representations with differing degrees of detail to show how the verb algorithm reports results. The unknown verb in each of the sentences is “throw”. The SNePSUL code for the “throw” demo can be found in Appendix B and the output in Appendix A.

Figure 1 is a representation of the sentence:

”Derek threw the baseball.”

Notice all the information that needs to be modeled in order for the verb algorithm to return a useful result on what at first appears to be a simple sentence. At this point no effort is made to classify the verb throw (as bitransitive, transitive, etc.)

After running Ehrlich’s verb algorithm on Figure 1, the output would is as follows:

(A (human) CAN THROW RESULT= NIL ENABLED BY= NIL)

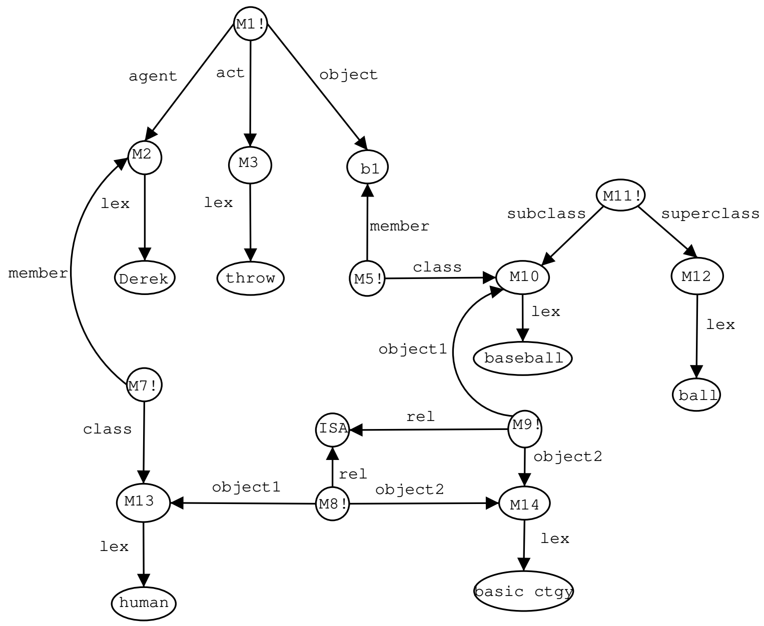


Figure 1: A representation where no verb type is specified.

Lost in this representation is the fact that "Derek" (the human) can throw a "ball". This is because "throw" was not labeled as being used in a transitive sense. As a result Ehrlich's algorithm returns information from the perspective that the verb is intransitive. This is the default case and the worst case scenario for the verb algorithm.

Figure 1 also uses a basic-level category to label both "baseball" and "ball". In order for the verb algorithm to return a descriptive type for the subject, object or indirect object the node at the end of each respective arc must be labeled a basic-level category or of class animal. The algorithm simply cannot take into account the infinite number of possibilities at the end of the arcs and uses these two categorizations to bring some order to the problem.

Figure 2 is a representation of the following sentence:

"Derek threw the baseball to Tino."

Here the verb is correctly labeled as bitransitive. This changes the results reported by the

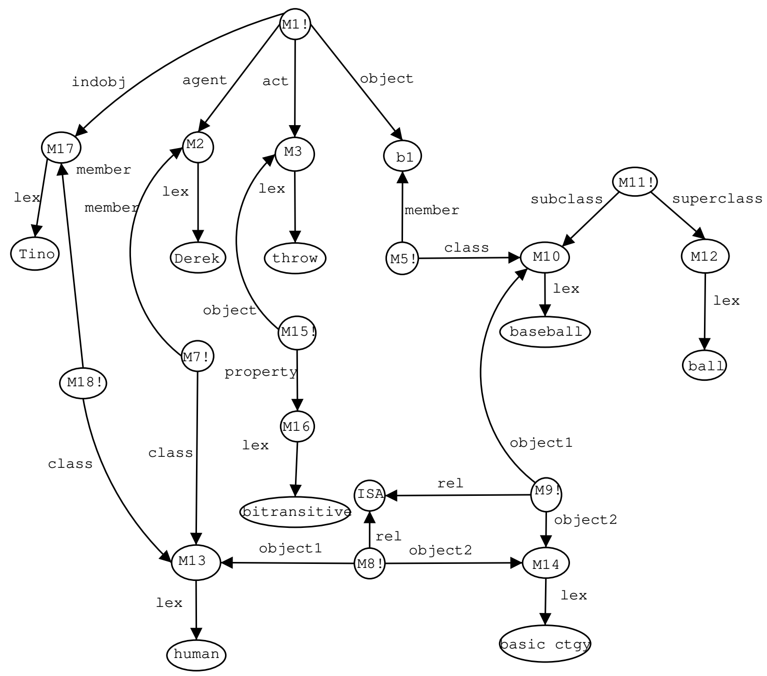


Figure 2: A representation where verb type is specified.

verb algorithm and after running Ehrlich's verb algorithm on Figure 2, the output would be as follows:

(A (human) CAN THROW A (baseball) TO A (human) RESULT= NIL ENABLED BY= NIL)

The results match the grammatical structure of the sentence because the verb is correctly labeled as being bitransitive. The verb algorithm is able to report information on the agent, act, object and indirect object.

Figure 3 is a representation of the following two sentences:

"Derek threw the baseball to Tino."

"Bobo throws the rubber ball."

Here another instance of "throw" is introduced into the overall representation. Note in

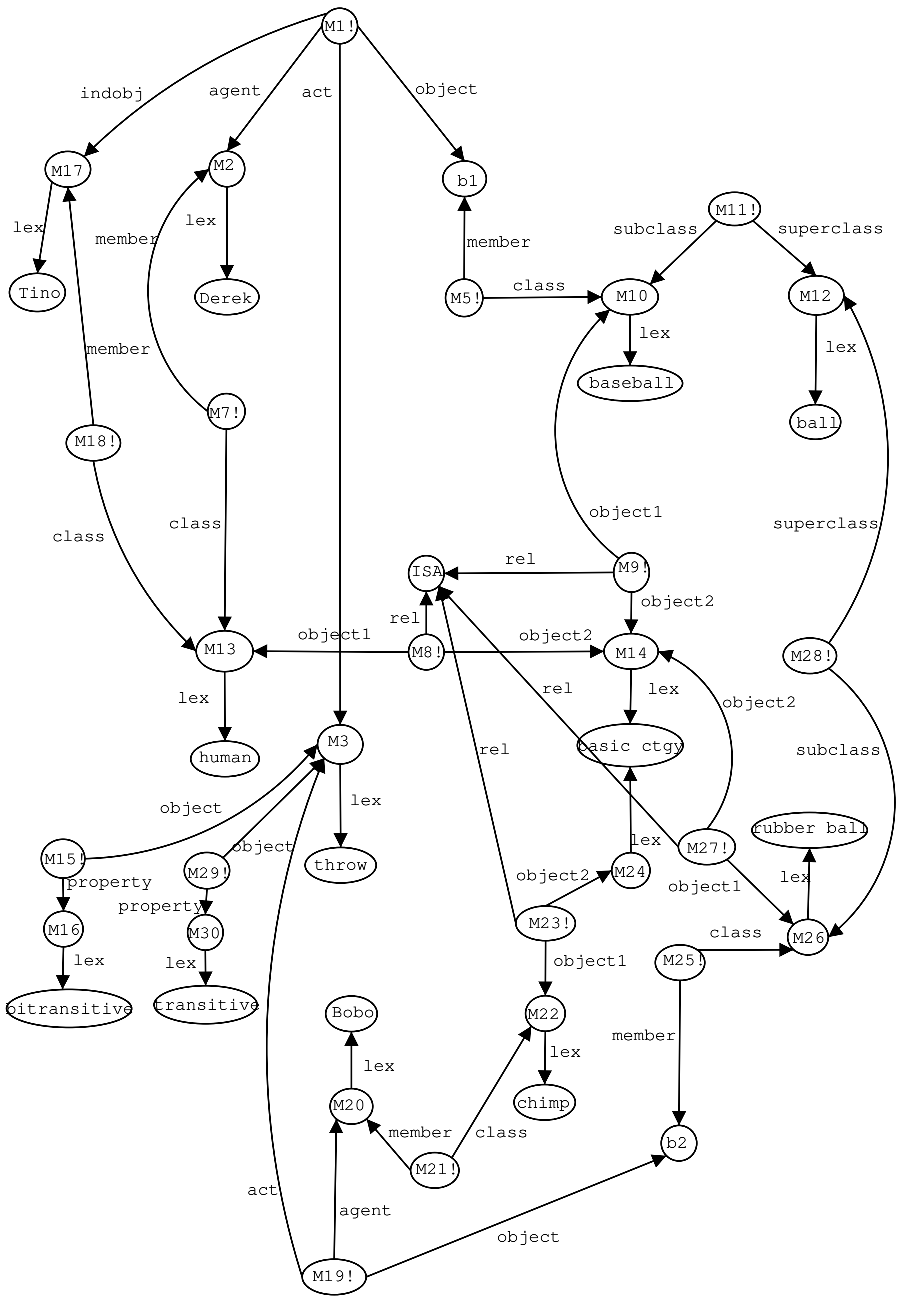


Figure 3: A representation with a verb being both transitive and bitransitive.

Figure 3 that "throw" is modeled as being both transitive and bitransitive as instances of each type exist. The verb algorithm will search on bitransitive first, stop upon finding a single instance and report all findings as if they are bitransitive in nature thereby ignoring entirely the transitive categorization that exists in the representation.

The output for the algorithm is:

(A (human chimp) CAN THROW A (baseball rubber ball) TO A (human)

RESULT= NIL ENABLED BY= NIL)

Notice that in the output no distinction is made between the two types present in the representation. Also note that it is not evident from the original two sentences that a human can throw a ball to a chimp but the output does not make this distinction clear. This is discussed further in Section 9.3.

4 Case Frames Used in Representations

Here is a list of the standard SNePS case frames used in the forthcoming "proliferate" passage and background representations:

lex	member/class	superclass/subclass
agent/object/act	object/act	object/property
mod/head	object1/rel/object2	object/rel/possessor
&ant/cq	ant/cq	

Formal definitions for these case frames have not been included because excellent definitions already exist. The reader is directed to Shapiro and Rapaport (1992-94) and Shapiro and Rapaport (1987) for in-depth definitions of most of the standard SNePS case frames listed above.

There are numerous case frames specific to Ehrlich's noun and verb algorithms. A complete description of the case frames specific to the noun and verb algorithm can be found in Ehrlich (1995: 74). Several figures in the report reference some of the non-standard case and brief explanations of each are included below.

members/class - This is a variant of member/class and is Ehrlich's way of modeling a collection of elements that are of basic-level. Consider the following SNePS representation:

```
(M1  (members B1)
     (class (M2 (lex racehorse))))
```

This representation states that node B1 is a collection of members of the class race horse. Ehrlich admits that this case frame is nothing but a temporary stop-gap until a more comprehensible case frame for collections is developed. The reason members/class is used at all in the proliferation representations is because modeling collections of elements that are basic-level allows the verb algorithm to return useful results. Standard case frames (like superclass/subclass) could have been employed to represent collections but they would not have functioned with the verb algorithm.

act/object/source - This is a variant of the agent/act/object type case frames. Here a source arc has been added to help define a specific type of sentence. Consider the following sentence:

The gold taken from the Incan empire.

A SNePS representation would be:

```
(M3  (act (M4 lex take))
     (object (M5 lex gold))
     (source (M6 lex Incan empire)))
```

The representation states that the gold (M5) was taken from a specific source, in this case the Incan empire (M6).

This of course is a very specific case frame and limited in its applications. Ehrlich tends to take the agent/act/object case frame and tag on additional arcs like place, time, manner, to, from, instr, indobj, location, etc. to fit a variety of different sentences. This allows Ehrlich to categorize almost any sentence but is in a hindrance in the sense that the number of potential additional arcs that could be added to an agent/act/object case frame is probably in the hundreds.

agent/act/object/location - This is a variant of the agent/act/object type case frames. Consider the sentence:

John walked the dog in the hall.

A SNePS representation would be:

```
(M7   (act (M8 lex walk))
      (agent (M9 lex John))
      (object (M10 lex dog))
      (location (M11 lex hall)))
```

The representation states that the dog (M10) was walked in a specific location, the hall (M11).

5 Proliferate Passage

Listed below is the passage that is the focus of this report. It is taken from a Newsweek article discussing scientific research using stem cells. The passage is a SMET type text and the word "proliferate" is one that has a decent change of being unknown by a middle school reader.

A decade ago research on lab animals revealed that stem cells taken from animal embryos are astoundingly versatile. They grow in the lab, **proliferate** like rabbits and turn into specialized cells such as neurons (Begley 2001: 25)

The most interesting aspect of the passage is the simile relation that exists between rabbits and stem cells expressed in the second sentence. The simile provides a host of different angles of attack that can lead to possible definitions of proliferate based on its context and accompanying background knowledge.

For reference purposes a common dictionary definition for proliferate is:

To produce, reproduce, or grow, especially with rapidity, as cells in tissue formation.

6 Representation of Passage

The passage can be broken into two obvious logical parts, the first and second sentence. From there both sentences can be broken down into clauses. The overall representation uses a modular approach first breaking the sentences into clauses, representing them as individual networks and then combining the networks back into complete sentences. It should be noted that the overall approach for the representation closely follows the work of Marc Broklawski (2001). It was extremely helpful to have Marc's original representations as they provided a great degree detail and thought. Changes to Marc's original representations come with the introduction of Ehrlich specific case frames and new rules.

It is important to note the difference between an absolutely perfect SNePS representation for the passage and a representation that is geared toward working with the verb algorithm. Davis, Shrobe and Szolovits (1993: 19) propose that knowledge representations inevitably bring "some part of the world into sharp focus at the expense of blurring other parts". It is therefore necessary to focus on the problem at hand, creating a representation that works with the verb algorithm, while knowingly avoiding a complex representation that takes into account numerous aspects that would provide a more complete representation but have no apparent bearing on the verb algorithm. Where Ehrlich specific case frames are used rather than standard case frames the fact is pointed out. At no time are representations badly forced or blatantly inaccurate for the simple goal of extracting information from the verb algorithm. If anything the representations serve as important evolutionary steps in an effort to extract new ideas as how to improve the verb algorithm and make it viable and robust.

6.1 Sentence One

It is unfortunate that this sentence lends only a small portion of information actually used by the verb algorithm but this should not detract from its importance. The sentence may provide bits of information that could be helpful in refining future definitions for "proliferate" when a more refined background knowledge representation exists.

The sentence itself can be broken into four subparts.

1.1 research on lab animals

1.2 stem cells taken from animal embryos

1.3 stem cells taken from animal embryos are astoundingly versatile

1.4 The research itself revealed that stem cells taken from animal embryos are astoundingly versatile.

Figure 4 shows subpart 1.1's representation. Here "lab animal" is modeled as a class and the actual set of lab animals used in the research referred to in the sentence is represented as a

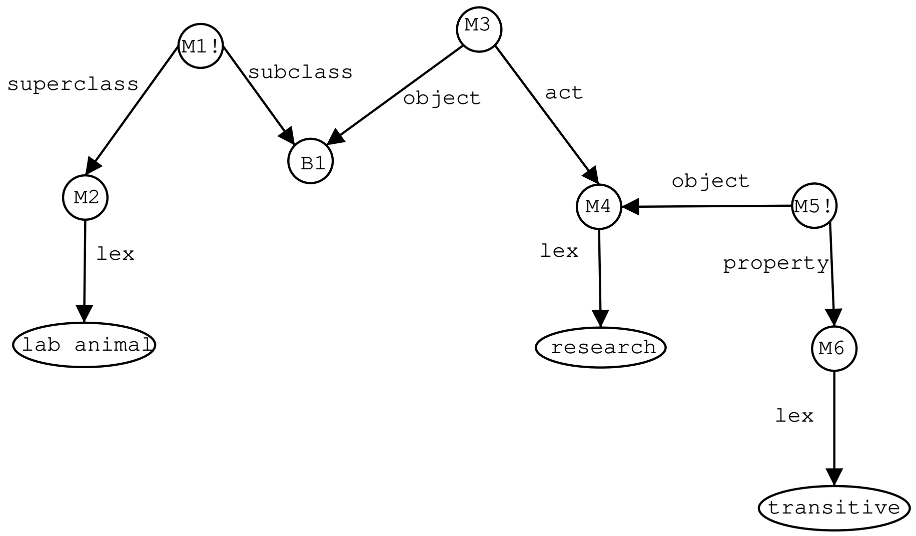


Figure 4: A representation of “research on lab animals”

subclass of the class “lab animal”. The lab animals are modeled as a base node (B1) because information regarding the type of lab animal, number of such animals, etc. is not given. Ehrlich may have differed in her representation of this collection and used objects1/are/object2 case frame as a way to represent it. Using the Ehrlich specific case frame provides no distinct advantage when running the verb algorithm so it is disregarded.

Figure 4 is also important because it shows the “research” labeled as a transitive verb based on its usage. The verb algorithm requires that whoever models the data (either the knowledge engineer or parser) apply a label to each verb based on its predicate structure. The algorithm looks for an object/property case frame where object point to the verb being searched and property point to one of the four verb types. Alternatives to hard-coding the verb type are touched on in Section 9.2.

Figure 5 shows subpart 1.2. The complex representation uses mod/head (M15) as its entry point. The phrase can be broken down into two logical parts: One, “stem cells”, and two, “taken from animal embryos”. The representation places “stem cells” in the head location and “taken

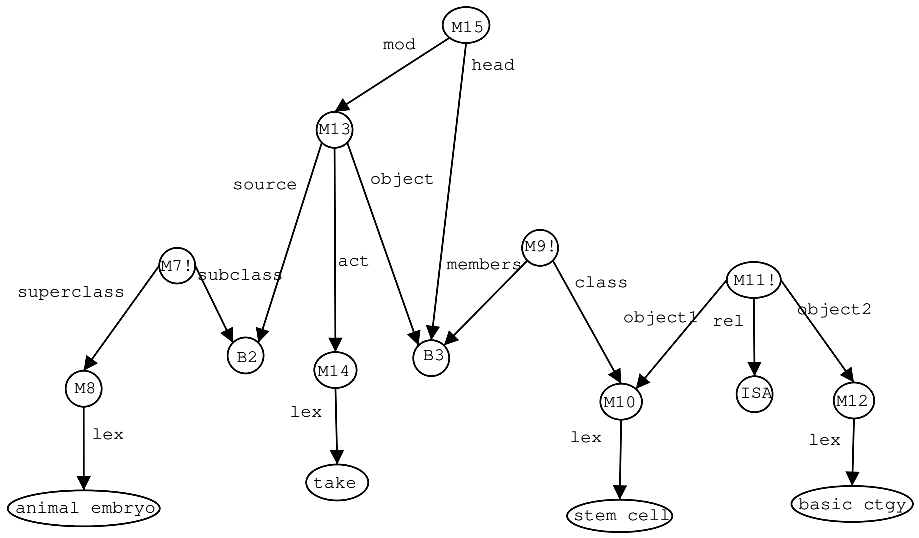


Figure 5: A representation of "stem cells taken from animal embryos"

from animal embryos" in the mod location. The mod/head case frame implies that the stem cells spoken about in the sentence have the quality of being "taken from animal embryos".

An important case frame introduced in Figure 5 is object1/rel/object2. The verb algorithm takes special note when agent, object or indirect object arcs point to a node that is itself also in an object1/rel/object2 relation with the rel arc pointing to "ISA" and object2 pointing to "basic ctgy". This is how Ehrlich defines basic-level categories. For the purposes of the representation labeling "stem cell" (B3) as a basic-level category is appropriate. It may be the case that for certain individuals the class "stem cell" would have many logical subclasses but it is safe to assume that for most readers it would be viewed as basic-level. This is a completely subjective categorization.

Other interesting points about Figure 5 are the base nodes themselves used to represent the stem cells and animal embryos. Much like the lab animals in subpart 1.1, little specific information is known about the stem cells or animal embryos in question (like their exact type, quantity, etc.), thus their modeling as base nodes.

Figure 6 is a representation of subpart 1.3. It shows that the "stem cells taken from animal

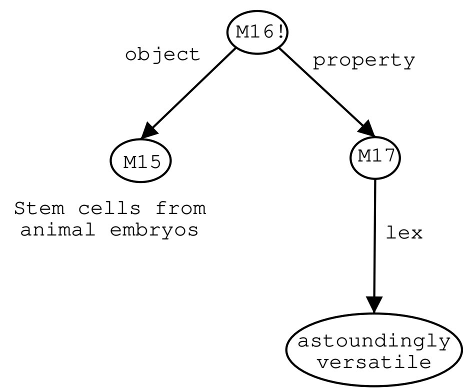


Figure 6: M15 has the property of being astoundingly versatile

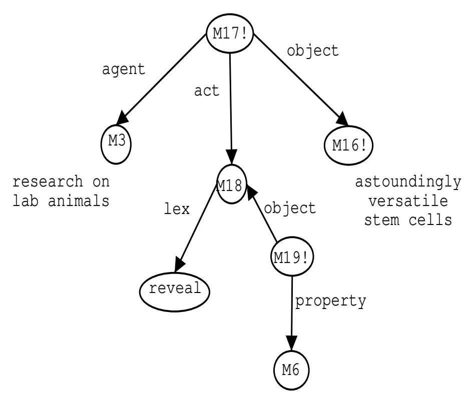


Figure 7: A complete representation of sentence one

embryos" have the property of being "astoundingly versatile".

Figure 7 is a complete representation of sentence one. The sentence is represented using the agent/act/object case frame. Note that M17! has an agent, act and object arc emanating from it. If the verb algorithm were run at this point on "reveal" the output would be:

(A NIL CAN REVEAL A NIL RESULT= NIL ENABLED BY= NIL))

Here results are returned that in some ways are misleading.[1] Ehrlich's verb algorithm

[1]All representations in this report attempt to define the type of each verb, not just the unknown verb "proliferate". This was done in attempt to gain as much information about the verb algorithm as possible. In many cases it yielded

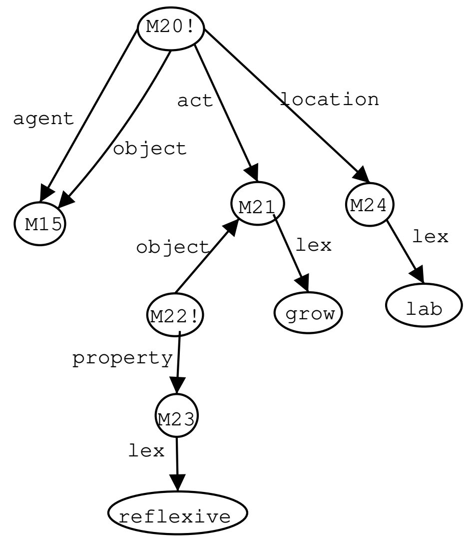


Figure 8: A representation of "They grow in the lab."

cannot deal with agent or object arcs that point to anything other than basic-level categories or instances of class animal. This is a perfect example of where reliance on basic-level categorization leads to the verb algorithm returning less information than it could.

6.2 Sentence Two

This sentence presented many more difficult challenges than sentence one because it uses a simile. Sentence two can be broken into the following three subparts:

2.1 "They grow in the lab"

2.2 "They proliferate like rabbits"

2.6 "They turn into specialized cells such as neurons"

Figure 8 is a representation of subpart 2.1. Grow is modeled as being reflexive in this instance and M15 is both the agent and the object of proposition. Node M15 refers to subpart

helpful bits of information that would not have been found otherwise.

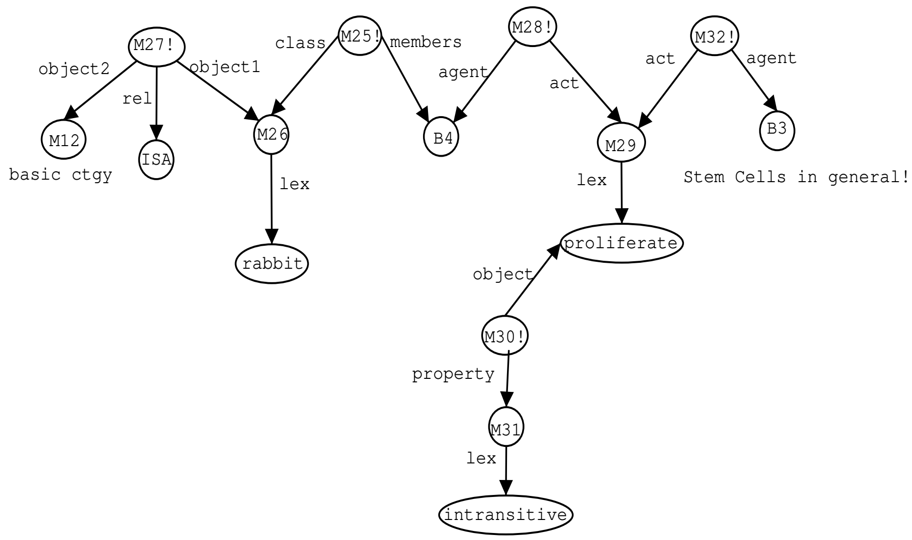


Figure 9: A representation of "They proliferate like rabbits."

1.2, "stem cells taken from animal embryos". Here it is assumed that actions are being done to the entity M15 and not simply stem cells which are a component of M15. This will create a slight problem in subpart 2.2 as the representation uses stem cells B3 (see Figure 5), which are basic-level, rather than M15 which is not a basic-level.

Figure 9 is part of the representation of subpart 2.2 and is straight-forward and easily understood. Node B4, a subset of the class "rabbit", is modeled as being able to do the action "proliferate". The class "rabbit" is additionally modeled as being a basic-level. Figure 9 models B3 (the collection of stem cells created in Figure 5) as being able to perform the action proliferate. This differs from Marc's original representation as he had "stem cells taken from animal embryos" being able to proliferate and not simply "stem cells". Although in some respects Marc's representation is more accurate and informative it fails to produce useful results for Ehrlich's verb algorithm. If node M15 ("stem cells taken from animal embryos") was to be substituted for B3 the verb algorithm would fail to return useful results as M15 is not a basic-level. The current representation is adequate for the problem at hand.

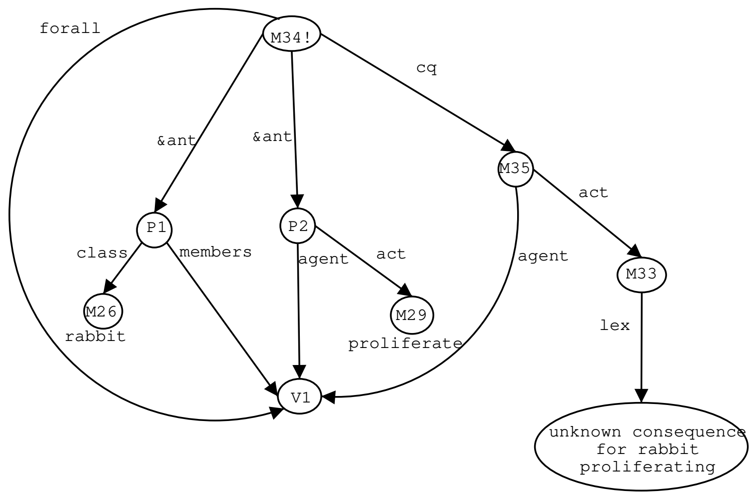


Figure 10: A representation of a consequence for rabbits proliferating.

Also note that "proliferate" is labeled as intransitive. This is not entirely necessary as intransitive is the default category for all verbs and had "proliferate" not been labeled the verb algorithm would have applied this category to it.

Subpart 2.2 leads to two issues that need to be addressed:

1. It would be nice if Ehrlich's verb algorithm reported some type of consequence for the verb proliferate. A human reader encountering the verb for the first time would more than likely model a consequence for the verb (whether it be right or wrong) in order to continue reading the passage.

2. An additional representation is needed to model the simile relationship that exists.

Creating a Consequence for "proliferate"

A consequence for a rabbit proliferating is modeled in Figure 10.[2] It uses the following rule to create the consequence:

> **Forall r**
>
> **If r is a rabbit and r proliferates**
>
> **then r does some unknown consequence**

Since proliferate's meaning is unknown the only consequence that can be modeled at this point is an unknown one. M35 models the unknown consequence and is the most important node in Figure 10. It may seem a waste of time to model an unknown consequence but it is indeed useful information for the verb algorithm to return to the user. At the very least it would inform any interested party that the verb proliferate is unknown to Cassie.

Representing the Simile

The next step is to use the object1/rel/object2 case frame to represent a simile relationship. Figure 11 is a representation of the simile in the second sentence.

First some theory on what a simile is and what are its necessary components that need to be modeled. A simile requires that: One, two items are compared and two, that the reader comprehend the context in which the comparison is made (Way, 1994: 38). For example consider the sentence "Juliet runs like a deer." The reader would first need to understand that Juliet can run and that a deer can run and that both of these actions are comparable based on some context. The interesting point is that the simile tells the reader the obvious part, that Juliet and deer run,

[2]In Appendix D an accompanying consequence is modeled for a stem cell proliferating. It has been left out here for brevity.

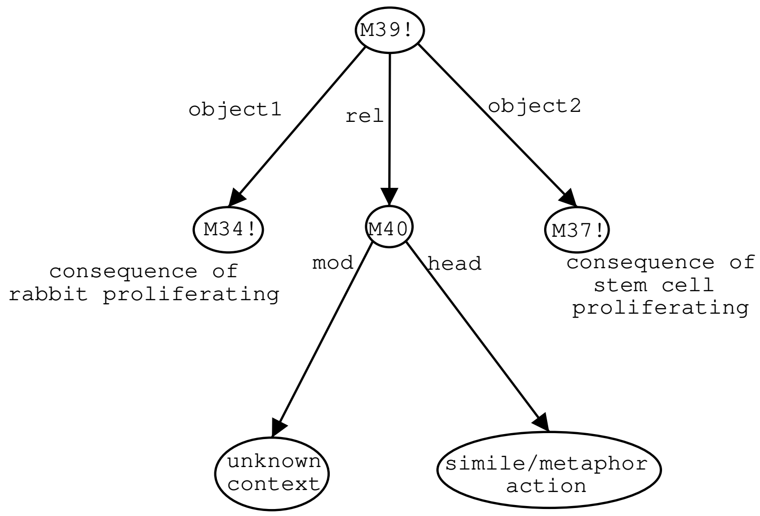


Figure 11: A representation of a simile relation

but neglects to model the difficult component; the context in which their running is similar. It is left to the reader to implicitly make the correct connection. But how does the reader know that the sentence “Juliet runs like a deer.” is supposed to be similar based on the context of ”gracefulness” and not for instance “quickness”? This is a very complex issue and is partially addressed in Section 7.4.

Figure 11 attempts to capture all of these required elements for representing a simile. It has the two items that are being compared, in this case “stem cells proliferating” and “rabbits proliferating”, it has the simile relationship and it has the context in which the comparison logically makes sense to the reader. Note node M40 has a head pointing to “simile/metaphor action” and a mod pointing to an “unknown context”. A literal English translation of M39! is: “A consequence of a rabbit proliferating is like a consequence of a stem cell proliferating based on some unknown context.”.

Having an object1/rel/object2 case frame for similes is unfortunately not enough and a rule needs to exist that gives meaning to this specific relation. This is covered by Figure 12. The rule

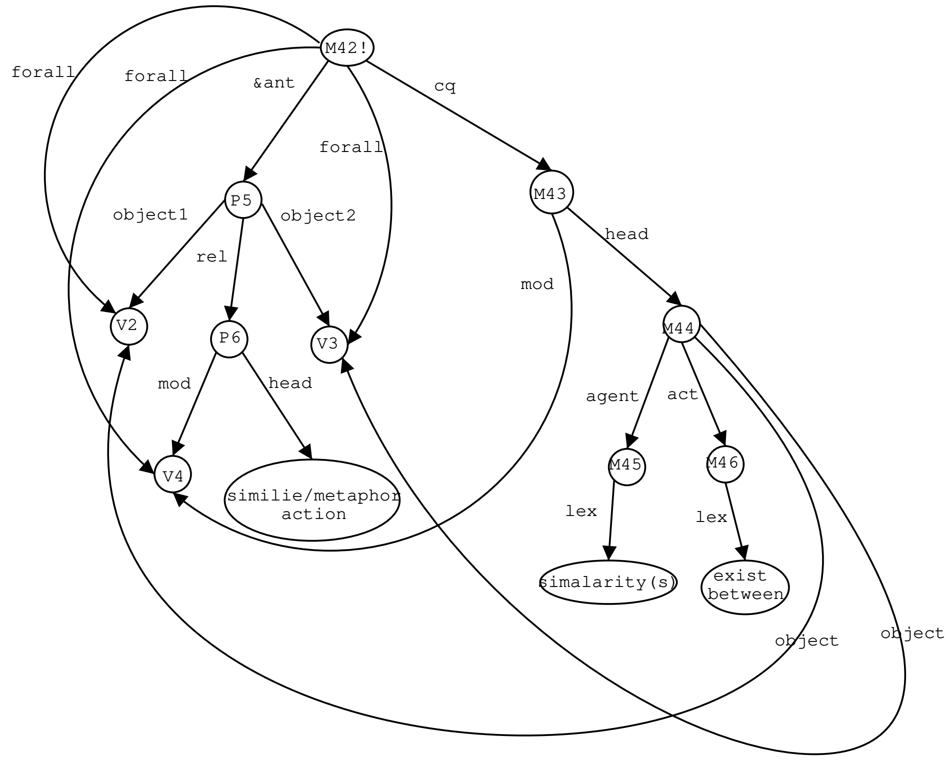


Figure 12: A rule for interpreting a simile relation

states exactly what it means for two items to be in a simile/metaphor action relationship based on some context and the literal translation is:

Forall all x, y and z

If consequence x and consequence y are in a "simile/metaphor action" relation modified by context z,

then similarities exist between consequence x and consequence y based on context z.

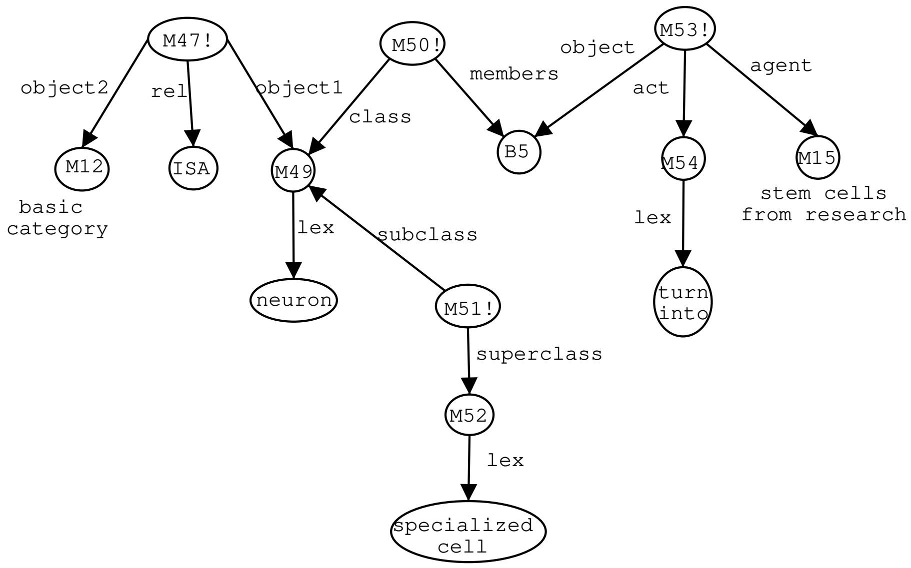


Figure 13: "They turn into specialized cells such as neurons."

Figure 13 shows the representation for Subpart 2.3. The sentence is "They turn into specialized cells such as neurons.". M50! represents the portion of text that reads "specialized cells such as neurons". The neurons in question are modeled as B5 and the fact that these neurons are specialized cells is represented by their being descended from the class specialized cell. At this time subpart 2.3 provides little or no information used by the verb algorithm.

6.3 Results of Verb Algorithm

The SNePSUL representation for Sections 6.1 and 6.2 are found in Appendix D and the results of the verb algorithm run using these representations as input are found in Appendix C. Below is what the verb algorithm returns as a result for proliferate:

(A (stem cell rabbit)

CAN PROLIFERATE RESULT= (P82 P78) ENABLED BY= NIL)

It states that a stem cell or rabbit (both of which are basic-level in the representation) can proliferate with the result P82 and P78. P82 and P78 are the consequences defined for proliferate for a rabbit and stem respectively. At this time P82 and P78 state there is some unknown consequence for proliferate. These results can hopefully be further refined by the introduction of background knowledge and this is the goal of Section 7.

7 Background Knowledge for Passage

In many ways creating an appropriate background knowledge network is a more difficult task than representing the passage itself. The knowledge engineer reviews the passage for clues and makes inferences as to what combination of textual clues and background knowledge an average reader would use to refine the original definition of the unknown word. The knowledge engineer contemplates what he has been given and creates lines of attacks, much like a chess game, that he believes are logical, reasonable and will yield useful results.

For the purposes of representing "proliferate" the primary question the knowledge engineer ponders is "What information does the simile relationship between both rabbits and stem cells provide that can be used to refine the meaning of the verb proliferate?". Below is set of steps, each building off the previous, that is used to create a background knowledge network capable of producing a refined definition for proliferate. This sequence of steps was a joint effort decided upon by the Justin Del Vecchio and Dr. Rapaport. The sequence of is not meant to be a template for all situations where an unknown verb is encounted and is customized for this specific instance of "proliferate".

Overview of Steps

- **Step One** - List the actions and characteristics that both rabbits and stem cells have. For instance a rabbit can eat, hop and reproduce. It also has the characteristics of being furry, cute and quick. A stem cell can do actions like divide, live and die. The idea here is to model all the actions and characteristics an average reader would be expected to know about stem cells and rabbits. After defining all the actions and characteristics, define a subset of these that both stem cells and rabbits have in common. For instance a stem cell and rabbit are both able to live and reproduce.

- **Step Two** - Explicitly state in the SNePS representation those actions which rabbits can do that stem cells cannot (and vice versa). For instance if a rabbit can hop and a stem cell cannot then a rule must exist that enforces the fact that stem cells do not have the ability to hop. This is a required step because although it might be blatantly obvious to a reader that a stem cell is unable to hop it is not obvious to Cassie unless it is a proposition in her mind.

- **Step Three** - Review all the actions that are common to both stem cells and rabbits and make a guess about which actions are most closely related to proliferate. This is clearly the most difficult step as the forthcoming representations attempt to model human thought along the lines "I had a hunch it might mean that.". Here the representations are scoured for sometimes subtle and other times obvious clues that relate to the reader that although stem cells and rabbits have a specific action in common the meaning of the action cannot be viable substitute for proliferate because it simply does not fit. For example both stem cells and rabbits die so the consequence of dying might be a possible consequence of proliferate. Upon re-reading the simile "They grow in the lab, proliferate like rabbits and turn into specialized cells such as neurons" (Begley 2001: 25) it is clear that the action dying does not fit for this simile. If anything the fact that the stem cells are growing clearly implies that they are happily living. To get a human to make this deduction is simple but to get Cassie to make this deduction requires coding in the information.

- **Step Four** - Now that a set of actions common to both stem cells and rabbits exist, the set defined in Step Three, the reader can ask further questions about the simile like "Why is a rabbit used in the simile and not a cat?" At this point a truly deductive reader would compare the actions and characteristics common to both cats and rabbits and find those actions and characteristics that differ. Is it possible that one of these differences, maybe the fact that rabbits have a very fast birth rate and cats do not, will provide further insight as to the meaning of proliferate? This is in some ways an even more difficult task than Step Three and is not touched on in this report.

The goal of these steps is to create a new consequence for the verb proliferate which Ehrlich's verb algorithm can report. This new consequence need not be entirely correct, rather it need only be a consequence that an average reader might be expected to deduce from the context of the passage and appropriate background knowledge.

7.1 Step One

Here the actions and characteristics for both stem cells and rabbits are modeled. The first item modeled is a class hierarchy that places both stem cells and rabbits into an abbreviated taxonomical structure. Figure 14 is the representation used. It has a root class named "animal" and leaf nodes for both "rabbit" and "stem cell". Originally actions and characteristics were to be assigned to the classes in the taxonomical structure. For instance the class "animal" would have the actions living, dying and reproducing, the class "mammal" would have the characteristics of being hairy

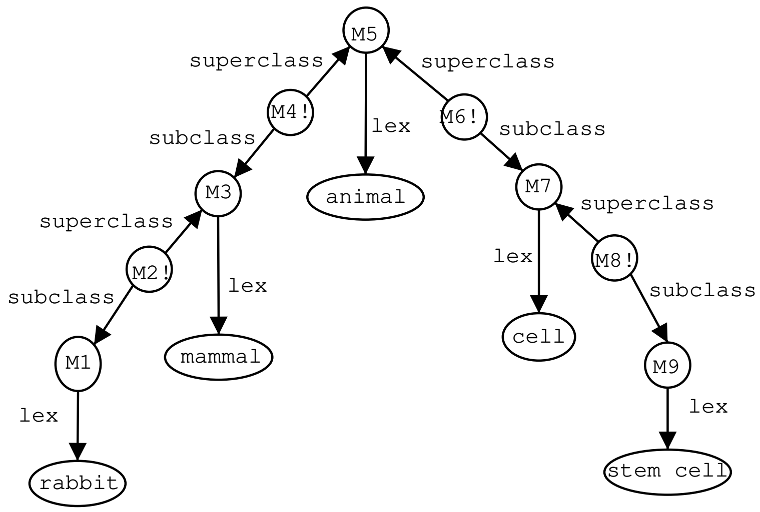


Figure 14: A hierarchy

and warm blooded and because “rabbit” descends from both of these it would have those actions by default. Unfortunately this hierarchy was not implemented because a proper path based inference mechanism could not be found that would not map the actions and characteristics for “stem cell” to the class “animal” and then to “rabbit”. Figure 14 is included because it provides more information for the verb algorithm to report on.

Figure 15 and Figure 16 show the characteristics of rabbits and items possessed by stem cells respectively. These two representations add little information for the verb algorithm to work with but are included because in the future it may be possible to create a rule that will review characteristics and generate a consequence for an unknown verb. Not a great deal of thought has been put into this approach but it certainly seems representations like these would be helpful if Step Four is to be implemented.

It was difficult to model characteristics for stem cells as they are not evident at first. Consider a stem cell, what characteristics does it really have? Possibly a shape, maybe a color but it is unfamiliar enough that distinguishing cell characteristics are not at first obvious. Instead

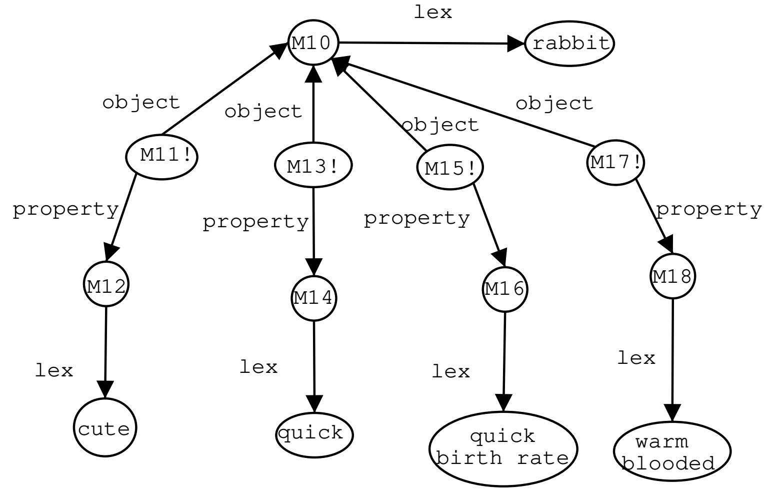


Figure 15: Rabbit characteristics

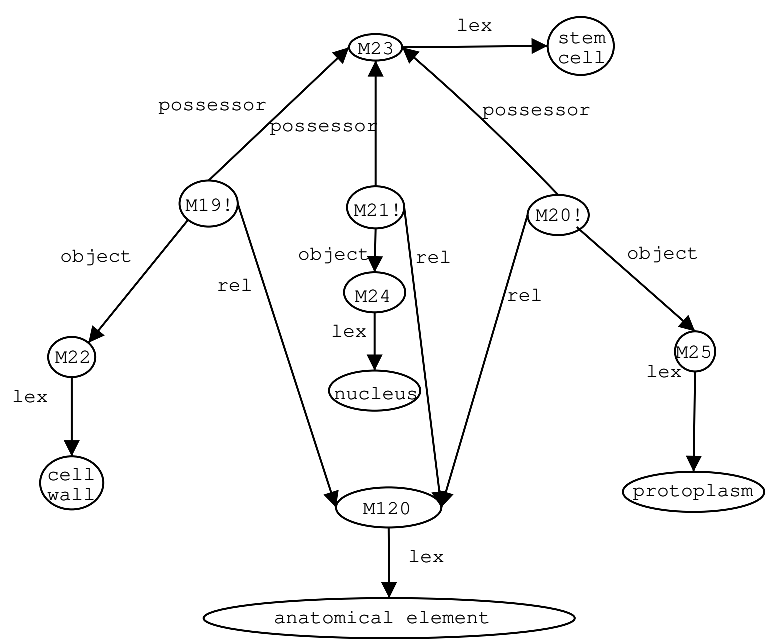


Figure 16: Things stem cells possess

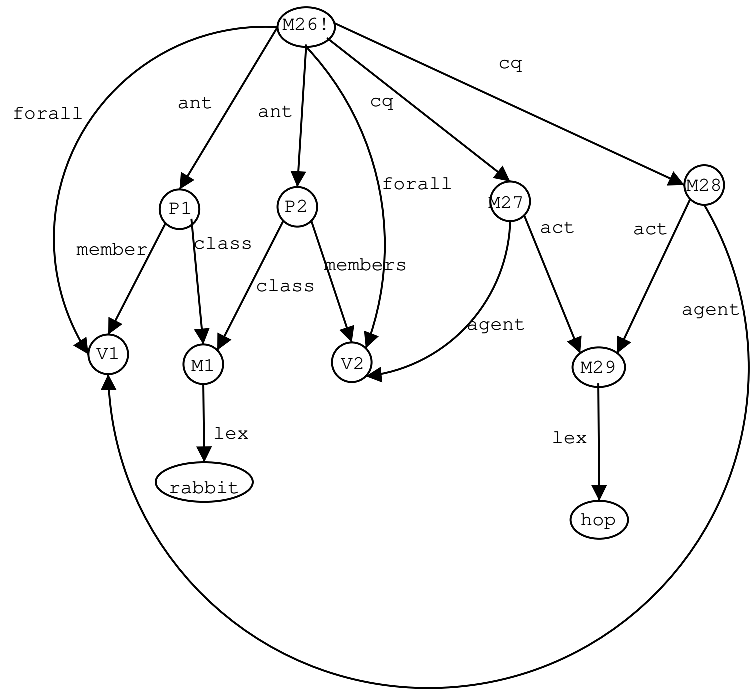


Figure 17: Rabbit actions

of creating characteristics Figure 16 models things that stem cells possess. This provides some information that might be used in the future.

Figure 17 shows the actions specific to rabbits.[3] For this representation rabbits hop and eat. Obviously these are not all the actions that rabbits are capable of but it was not evident how many actions needed to be represented (probably the more the better). A figure showing actions specific to stem cells has not been included because it would be a structural duplicate of Figure 17. The demo file in Appendix F includes SNePSUL code that stem cells can do cell division.

Note that in Figure 17 it was necessary to model Ehrlich's concept of collections because earlier representations involving rabbits used the members/class case frame. Future modelers should keep in mind Ehrlich's case frame for collections and make a decision early on whether to use or

[3]Figures may not show all actions actually modeled in the demo file included in Appendix F because it is difficult to keep the SNePSUL code in sync with the figures. Some figures may be lacking complete detail.

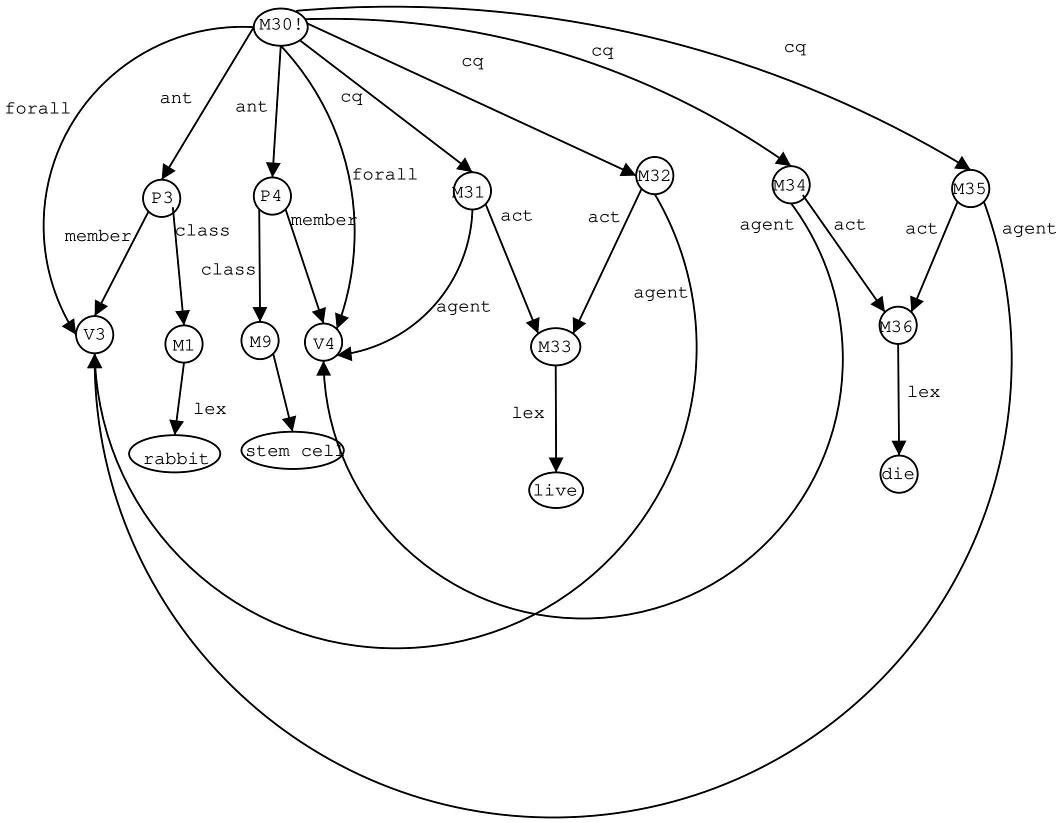


Figure 18: Actions common to both stem cells and rabbits

avoid it. If the verb algorithm's reliance on members/class is to be removed entirely it will require rewriting sections of the algorithm.

Figure 18 shows the actions that are common to both rabbits and stem cells. They include living, dying and reproducing. These actions are essential for creating an enhanced definition of proliferate as it is known that both rabbits and stem cells proliferate so actions that are common to both classes may yield a suitable consequence for proliferate. It just so happens that reproduce holds a consequence that fits nicely with proliferate, though neither live or die hold consequences of any use.

The rule used to assign these common actions is:

Forall x and y

If x or y are of class Rabbit and Stem Cell respectively,

then x or y can live, die or reproduce.

7.2 Step Two

Step Two requires that actions specific to a class are modeled as such. It is insufficient to tell Cassie that a rabbit can hop and hope it will infer that since only rabbits can hop then stem cells cannot. If a user were to program in the proposition that rabbits are able to hop and then ask Cassie "Can a stem cells hop?" she would reply with something along the lines "I do not know.". A user might expect Cassie to reply "No, stem cells cannot hop." but this goes against the open world view that Cassie has. If there is no proposition specifically stating that stem cells cannot hop then Cassie cannot infer this fact.

This leads to a very interesting situation. It is tempting to first model everything that stem cells can do and everything rabbits can do and then forget about the complement to these propositions; everything stem cells and rabbits cannot do. Due to time constraints this information has not been modeled but it certainly should be in the future.

7.3 Step Three

This step finds all actions that stem cells and rabbits have in common and gathers up the consequences for these actions. The goal is to create a suite of consequences based on these mutual actions and assign them as possible consequences for the verb proliferate. Next, the list of consequences for proliferate are re-evaluated and poor candidates are discarded. This review of consequences in order to cull out poor choices has not been implemented due to time constraints.

Stem cells and rabbits have three actions in common: living, dying and reproducing (At

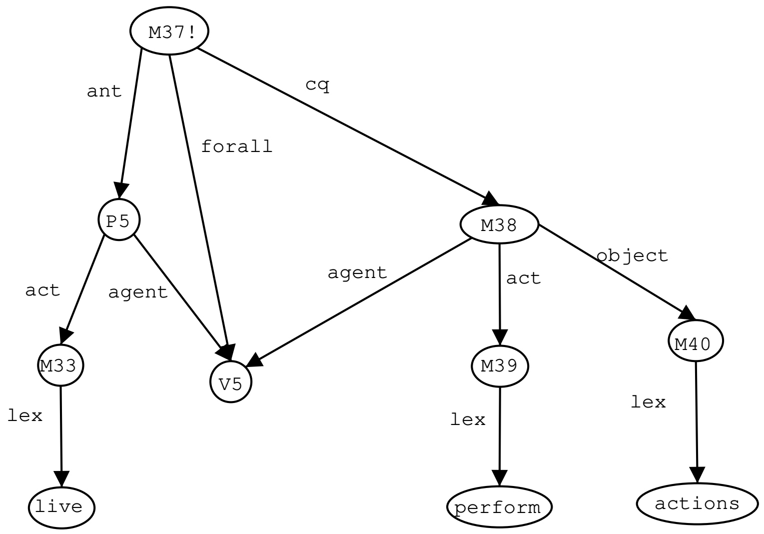


Figure 19: A consequence for living

least at first glance. There of course are many more but these three are evident without much thought). There is a possibility that a consequence for one of the common actions may also be an applicable consequence for proliferate. Therefore it is necessary to model the consequences for living, dying and reproducing. This is done in Figure 19, Figure 20 and Figure 21 respectively.

Assigning these consequences to proliferate without first reviewing their applicability is a debatable decision. It is questionable if humans would operate in this manner or if they would instead first rule out poor possible consequences and never bother assigning them. For the purposes of working with a SNePS representation it is much easier to assign a pool of possible consequences first and then root through them to find poor choices that can be discarded.

Figure 22, Figure 23 and Figure 24 show the hard coding of the consequences for living, dying and reproducing to the verb proliferate. Ideally this should be done by using a rule that basically states: "If actions are common to both rabbits and stem cells, then the consequences of these actions should be applied to proliferation.". This rule was not implemented due to time constraints.

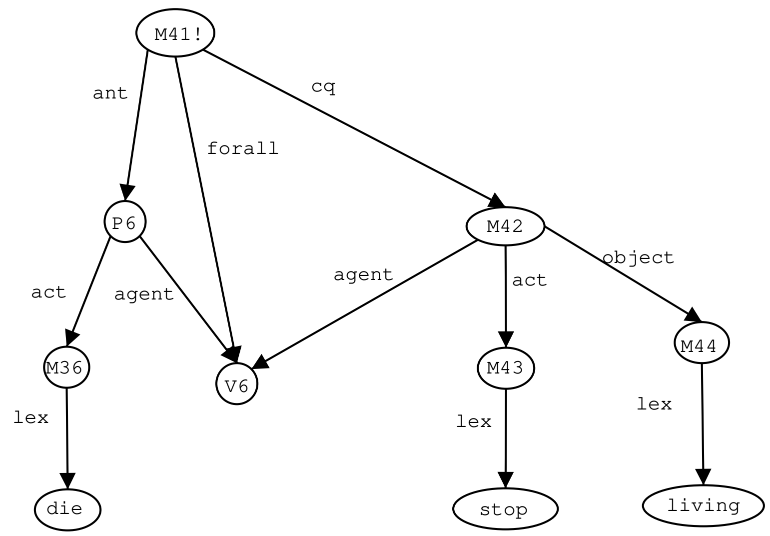


Figure 20: A consequence for dying

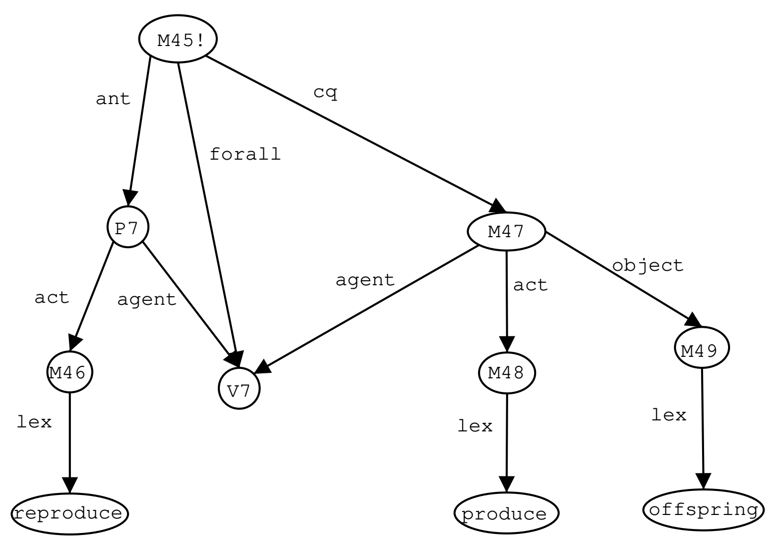


Figure 21: A consequence for reproducing

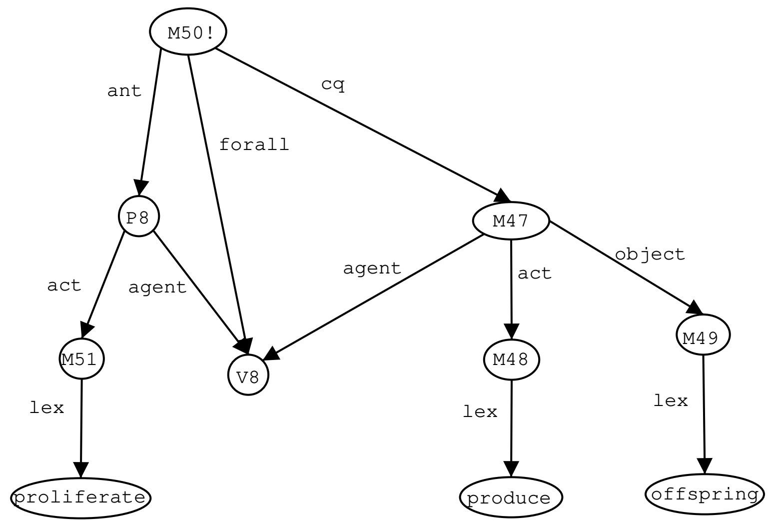


Figure 22: Linking the consequence for living with proliferating

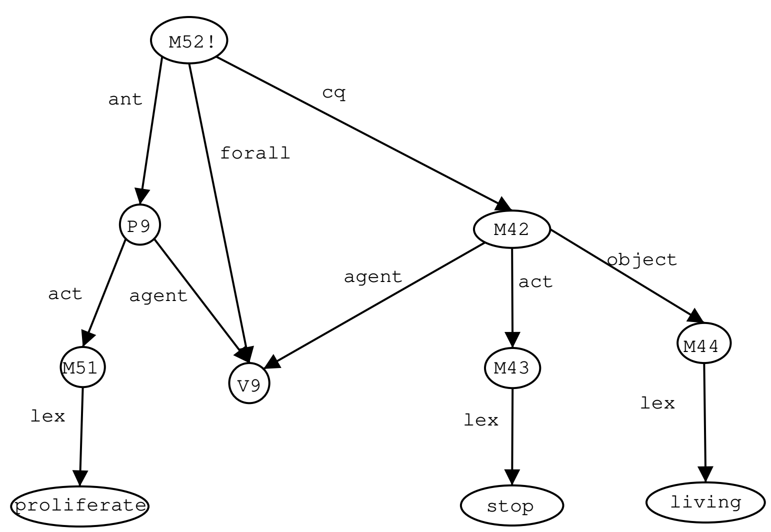


Figure 23: Linking the consequence for dying with proliferating

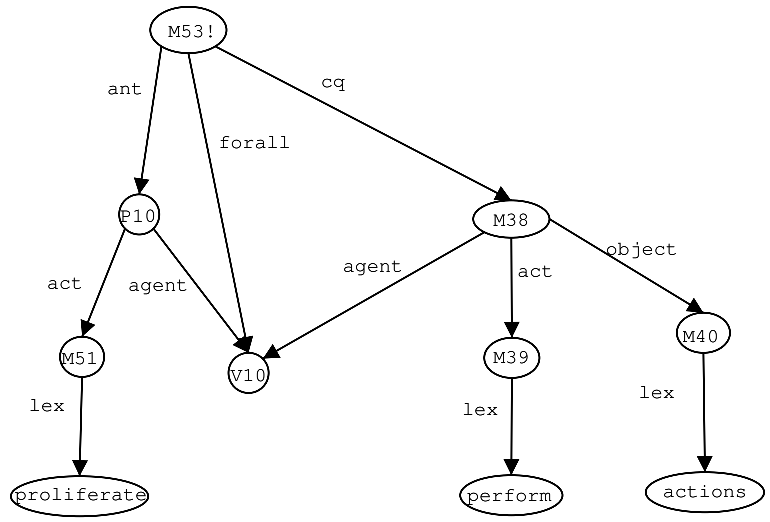


Figure 24: Linking the consequence for reproducing with proliferating

In general Step Three seems like a great deal of work in order to get the verb algorithm to recognize new set of consequences. At the very least the verb proliferate now has a consequence, the one for reproduction, that is a partial definition. However it also contains four other consequences which are of little use and in some cases blatantly incorrect.

It is now time to go about reviewing all the consequences and removing those that are inaccurate. But how should this be done? This would require modeling all the subtle steps a human takes when reasoning about a piece of text. A clue for the revision process is found in the context of the second sentence itself; "They grow in the lab, proliferate like rabbits and turn into specialized cells such as neurons." As mentioned earlier it obvious that this sentence implies the stem cells in question are alive. Therefore the consequence for dying would be a good candidate for removal from proliferate's new consequence list. This would require Cassie to infer from the sentence that she is dealing with live stem cells, to infer that if something is alive it is not dead and then to remove the dying consequence from proliferate. Encoding representations for all this information would take a considerable amount of time and effort.

7.4 Step Four

Step four is not included in any of the representations. It does further refinement by examining the simile relationship again but from a slightly different and more tricky angle than Step Three. Consider that rabbit is in a relation with a stem cell based on the action proliferate. It would be possible to compare something reasonably close with a rabbit, say for instance a cat, and deduce those properties which are different between a rabbit and a cat. The idea here is to figure out why a rabbit was used for the simile and not a cat. Based upon those differences it might be possible to deduce that rabbits reproduce quickly, cats do not and therefore any consequence for proliferate that has to do with reproduction has a higher likelihood than the rest to be the correct one. This task would require a great amount of detail.

7.5 Output of Verb Algorithm with Background Knowledge

Running the verb algorithm using the passage representations from Section 6 and accompanying background knowledge produces the following output:

(A (stem cell animal rabbit mammal cell) CAN PROLIFERATE RESULT
= (P180 P178 P176 P82 P78) ENABLED BY= NIL)

Note that the introduction of a taxonomical structure has allowed Cassie to infer that not only rabbits and stem cells can proliferate but also animals, mammals and cells. The results include three new propositions: P180, P178 and P176. The three new consequences are the consequences for living, dying and reproducing. Future work will require Cassie to evaluate the list of consequences and eliminate some obvious poor choices.

8 Future Work for Representation

The representation of the passage is fairly complete. Both sentences are represented and both use the Ehrlich specific case frames necessary for the verb algorithm to report useful information. At the current time it would be advisable for anyone picking up this work to review the passage's representation but avoid making major changes until a direction for the background knowledge is decided upon. A majority of the future work lies in creating a more solid and robust background knowledge representation. A set of possible steps would be:

1. Create a more encompassing background definition for both stem cells and rabbits. As it currently stands the definition lacks a comprehensive list of characteristics and actions for both stem cells and rabbits. Of course a list of this nature could reach an indeterminate size but the more information modeled the better. Additionally certain representations in the background knowledge need further review, specifically Section 7.3 should be reviewed to make certain the representations are accurate and fulfill the objective for which they were created.

2. Make certain that Cassie understands that actions and characteristics only rabbits have or can do stem cells cannot have or do (and vice versa). This seems tedious and not worth while but it is vitally important that Cassie have a complete definition that models the facts a human reader would know. Future work should not avoid this step.

3. Select a line of attack to further refine the definition of proliferate. Section 7.4 outlines one such avenue. This task will require a great deal of thought before actually coding up a representation. It easy to imagine that a decision made with respect to Step Four will take weeks if not months of time to code, test and implement successfully.

4. Begin looking at new passages and select one that will further help in refining the meaning of proliferate. While doing this exercise consider the background knowledge that will be required to implement the passage and choose one that appears to provide the most opportunities for a background knowledge that the verb algorithm can draw upon for information.

9 Suggestions for Improvements to Verb Algorithm

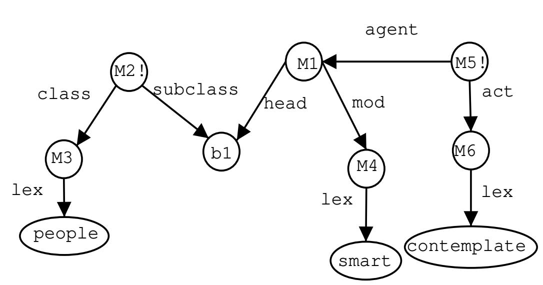
While working with the verb algorithm I came across three separate areas that I believe require further investigation and may eventually result in changes being made to the verb algorithm. I have ranked the suggestions in the order I believe they should be changed, an ordering I determined partly on necessity and partly on the time it would take to implement the change.

9.1 Modifying Basic-Level Reliance

Consider the following sentence:

"Smart people contemplate."

and its SNePS representation:



When the verb algorithm is run on this representation it responds with the result:

A SOMETHING CAN contemplate with result() or enable ()

It fails to report a more detailed description of the agent because the verb algorithm can provide English translations for only two categories:

1. Anything that an agent, object or indirect object arc points to that is modeled as basic-level.

2. Anything that an agent, object or indirect object arc points to that is of class animal in which case it is reported as being an "animal".

All other instances are reported as being "SOMETHING" (or possibly NIL if an empty list is returned).

This is somewhat disconcerting because there is a large difference between knowing a "smart person can contemplate" as opposed to "something can contemplate". I suspect that in order to solve this problem the algorithm should be changed to do the following:

1 - Find the act, object or indirect object arcs and then either:

2A - Have the verb algorithm examine the structure and return an appropriate English translation. This option was discussed in the April 30th lecture and Dr. Rapaport pointed out that this is the ideal solution. The parser and generator would be equipped with a finite set of case frames and as long as input representations adhered to these case frames, meaningful results could be returned.

2B - Ask whatever is at the end of these arcs to give back an English translation its representation. This requires each item that can be at the end of an act/object/indobj arc implement a lex arc that returns its English translation. The algorithm would have to be modified to search for and return the lex arc. Additionally the knowledge engineer inputting the information would have to remember to include it.

In the long term 2A is the obvious solution. In the short term 2B is the step that should be implemented. It will allow the verb algorithm to return more intelligible results. The amount of modification necessary to the verb algorithm is minimal while the benefits of making the switch are significant.

9.2 Categorization of Verbs

As mentioned throughout the report the verb algorithm relies on a predetermined categorization system for verbs. The knowledge engineer or parser must label an instance of the verb as being either bitransitive, transitive, reflexive or intransitive when the representation is created. The verb algorithm then searches for the verb's type and returns information based on the type found. There are two major problems with this approach.

First it is possible that verbs will be placed in incorrect categories or placed in no category at all. This may be less possible when it comes to a parser but quite possible when a human is entering the data.

Second, it is obvious that the actual structure of the representation, not an external categorization, should dictate which results are returned. For instance the verb algorithm should search for an act arc and then search for any accompanying agent, object and indirect objects (maybe even others) arcs rather then searching for its category and expecting a certain set of arcs to exist.

Implementing this change in the verb algorithm would be a difficult and time consuming task. It would require scrapping the report_bitransitive, report_transitive, report_reflexive and report_intransitive functions and creating a single, unified functions that would evaluate a preposition's structure and return the results dynamically. This is definitely a worthwhile task.

9.3 Verbs Should be Reported Instance Specific

Consider the following three sentences and assume that representations exist for each:

"Derek threw the baseball to Tino."

"The chimp threw the ball."

"Godzilla threw the metro bus"

The reporting mechanism for the verb algorithm would return the following output:

A (human chimp Godzilla) can throw a (baseball ball metro bus) to a human.

This leaves a somewhat unclear representation of who can throw what. It is likely that both a chimp and a human can throw a ball and baseball but unlikely that either can throw a metro bus. This is a delicate situation because it would be nice if the verb algorithm could report that both humans and chimps can throw balls and baseballs but it should not imply that they can throw a metro bus.

Getting the verb algorithm to separate each of the results so it would return "a human can throw a baseball to a human, a chimp can throw a ball to a chimp and Godzilla can throw a metro bus" is easy. It would be more complicated to have the verb algorithm return a representation " A human can throw a ball or baseball to a human, a chimp can throw a ball or baseball and Godzilla can throw a metro bus". The latter representation would require the verb algorithm to work extensively with the background knowledge representation for the passage. This problem requires further thought and consideration because the type of format finally decided upon will dictate the amount of time it takes to implement the change.

10 Conclusion

This project had three objectives: First, figure out what Ehrlich's verb algorithm does and the case frames it uses, second, represent Marc's original "proliferate" representations using Ehrlich's case frames with the goal of creating new representations that the verb algorithm can analyze and third, introduce appropriate background knowledge so the verb algorithm can return even better results for "proliferate". In order to understand Ehrlich's verb algorithm I created a SNePS demonstration, the "throw" demo, that tested the algorithm and showed its behavior in different circumstances. This helped immensely in understanding what was previously an unknown and in some ways undocumented entity. I was able to create new "proliferate" representations that worked well with the verb algorithm based on my findings from creating the "throw" demo and Marc's detailed original representations.

Representing the background knowledge was a greater challenge than first anticipated. It required coming up with a set of steps that detailed how a human might reason about a verb's definition based on its context and his background knowledge. The set of steps would then have to be implemented in Cassie in the form of background knowledge. I was able to create a background network that filled out most of these steps and provided the verb algorithm with more information to report. The "proliferate" background representations are a good baseline but far from complete.

My advice for future people working with the verb algorithm is to start considering the background knowledge immediately after reviewing prospective passages as the verb algorithm will rely heavily on the background network. Write down every possible observation that can be made about the background knowledge because no piece is too small or inconsequential to be important. It is only when all the potential pieces of background knowledge lay in front of someone that he can begin to rearrange information and see how it needs to be modeled in order to create a useful network. A common practice among object oriented programmers and screen writers is to write their ideas on 3 X 5 cards, place them on a table and begin rearranging them until they see patterns arise that are useful. I think this would have been a useful approach to modeling the background knowledge.

The verb algorithm is easy to understand and creates a useful definition. It provides knowledge engineers a great deal of flexibility when it comes to creating representation, especially background knowledge representations. The amount of time, effort and thought put into the SNePS representations to be analyzed by the verb algorithm directly translates to how accurate a definition the algorithm generates.

References

[1] Begley, Sharon(2001, July 9), "Cellular Divide", Newsweek: 23-31.

[2] Broklawski, Marc K. (2001), "Contextual Vocabulary Acquisition: Case Study of 'proliferate"' : 8-35.

[3] Davis, Randall; Shrobe, Howard; & Szolovits,Peter (1993),"What Is a Knowledge Representation?", *AI Magazine* 14(1): 17-33.

[4] Ehrlich, Karen A. (1995), *Automatic Vocabulary Expansion Through Narrative Context* (Buffalo:Department of Computer Science and Engineering of State University of New York): 74-80 and 96-108.

[5] Nagy, William E.; Herman, Patricia A.; & Anderson, Richard C. (1985), "Learning Words from Context", *Reading Research Quarterly* 20(2): 233-253.

[6] 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.

[7] Rapaport, William J., & Kibby, Michael W. (2000), "Contextual Vocabulary Acquisition: Development of a Computational Theory and Educational Curriculum", NSF grant proposal.

[8] Shapiro, Stuart C.; Rapaport, William J.; et. al (1992-94), *A Dictionary of SNePS Case Frames*(Buffalo: SNePS Research Group): 3-35.

[9] Shapiro, Stuart C.; Rapaport, William J. (1987), "SNePS Considered as a Fully Intensional Propositional Semantic Network", in Nick Cercone & Gordon McCalla (eds.), *The Knowledge Frontier: Essays in the Representation of Knowledge*(New York:Springer-Verlag): 277-288.

[10] Sternberg, Robert J. (1987), "Most Vocabulary is Learned from Context," in Margaret G. McKeown & Mary E. Curtis (eds.), *The Nature of Vocabulary Acquisition* (Hillsdale, NJ: Lawrence Erlbaum Associates): 89-105.

[11] Way, Eileen C. (1994), "Knowledge Representation and Metaphor" (Oxford:Intellect, Limited: 38-40).

Appendix A

SNePS output for the "throw" demo.

```
   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.


   5/4/2002 13:31:40


* (demo "throw.demo")



File #p/doc/graduate/663/demos/throw.demo is now the source of input.



 CPU time : 0.00


* ;;; Reset the network

(resetnet t)


Net reset



 CPU time : 0.00


*


;;; 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
    POSSESOR)

 CPU time : 0.01

*

End of file rels
```

```
 CPU time : 0.02


*

;;; 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 : 0.00


*

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



 CPU time : 0.00


*

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

```
                                   SUPERCLASS OBJECT1)

 CPU time : 0.00

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

 CPU time : 0.00

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

 CPU time : 0.00

*

End of file paths

 CPU time : 0.01

*

;;; Load Noun/Verb Algorithm
^(
--> load "code")
Loading code
```

```
Finished loading code

T




 CPU time : 0.30


*


;; 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 : 0.01

*

;; 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 : 0.09

*

(resetnet)

Net reset - Relations and paths are still defined


 CPU time : 0.00

*

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;; 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 : 0.04

*

;; 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 THROW A
   (baseball) RESULT= NIL ENABLED BY= NIL)



 CPU time : 0.21

*

(resetnet)
```

```
Net reset - Relations and paths are still defined



 CPU time : 0.00


*



;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;; 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 : 0.00


*
```

```
;; 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 : 0.00
```

```
*

;; 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 "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 : 0.27


*


(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 : 0.00


*


;; 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 : 0.00

*
```

```
;; 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 THROW A
   (baseball) TO A
   (human) RESULT= NIL ENABLED BY= NIL)



 CPU time : 0.50

*

(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 : 0.01


*


;; 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 THROW A

   (baseball

    rubber ball)

   TO A

   (human) RESULT= NIL ENABLED BY= NIL)





 CPU time : 0.60


*
```

```
(resetnet)

Net reset - Relations and paths are still defined

 CPU time : 0.00

*

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;; 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 : 0.00

*

;; 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

*

;; "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 : 0.05

*

;; 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 : 0.60

*

(resetnet)

Net reset - Relations and paths are still defined


 CPU time : 0.00

*
```

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;; 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 : 0.01

*

;; 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


*


;; 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 : 0.01

*

;; 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 : 0.01

*

;; 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 : 0.00

*

;; 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 : 0.88

*


(resetnet)

Net reset - Relations and paths are still defined
```

```
 CPU time : 0.00

*

End of #p/doc/graduate/663/demos/throw.demo demonstration.

 CPU time : 3.80

*
```

Appendix B

Here is the "throw" demo source code. It contains seven examples each of which are labeled "Example One", "Example Two", etc. The reader is directed to read the comments in order to understand the purpose of each example.

```
;;; 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
^(load "code")

;; 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)
```

Appendix C

SNePS output for the "proliferate" passage.

```
>

   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.

   5/2/2002 21:03:27

*

File #p/doc/graduate/663/demos/proliferate.demo is now the source of input.

 CPU time : 0.00

* ;; A representation of:
;; "A decade ago research on lab animals revealed that stem cells taken from
;;  animal embryos are astoundingly versatile.  They grow in the lab, proliferate
;;  like rabbits and turn into specialized cells such as neurons (Begley 2001)."
;;
;; Representations are based on Marc Broklawski's original representations though
;; changes have been made to adhere to the case frames used by Ehrlich's
;; algorithm.  Also new rules have been introduced.

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;  Setup work.  Define external paths and relations and load verb algorithm
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
```

```
;;; Reset the network

(resetnet t)

Net reset

 CPU time : 0.01

*

;; Define the relations we will need.

(define agent act object lext property objects1 object1 rel object2 ant cq &cq members class mod head source target transference vehicle forall &ant context arg1 arg2)ACT is already defined.

OBJECT1 is already defined.

OBJECT2 is already defined.

ANT is already defined.

CQ is already defined.

FORALL is already defined.

&ANT is already defined.

(AGENT ACT OBJECT LEXT PROPERTY OBJECTS1 OBJECT1 REL OBJECT2 ANT CQ &CQ
       MEMBERS CLASS MOD HEAD SOURCE TARGET TRANSFERENCE VEHICLE FORALL
       &ANT CONTEXT ARG1 ARG2)

 CPU time : 0.00

*

;;; 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.

AGENT is already defined.

CLASS is already defined.

EFFECT is already defined.

MEMBERS is already defined.

OBJECT is already defined.

OBJECT1 is already defined.

OBJECTS1 is already defined.

OBJECT2 is already defined.

PROPERTY is already defined.

REL 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
    POSSESOR)

 CPU time : 0.02

*
```

```
End of file rels



 CPU time : 0.02


*


;;; 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 : 0.00


*

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



 CPU time : 0.00


*
```

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



 CPU time : 0.00


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



 CPU time : 0.03


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



 CPU time : 0.01


*

End of file paths



 CPU time : 0.05


*

;;; Load Noun/Verb Algorithm
```

```
^(
--> load "code")
Loading code
Finished loading code
T



 CPU time : 0.26


*


;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;  Representation of:
;;  "research on lab animals"
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

;; Model the subclass of lab animals upon which research was performed.  A
;; subclass/superclass case frmae is used.
;; Ehrlich might not have used a subclass/superclass to represent this set.  She
;; instead might have done the following:
;;
;; object1 #setOfLabAnimals
;; rel "ARE"
;; object2  (build lex "lab animal")
;;
;; a variant which she admits is far from ideal but might be
;; needed for the purposes of her algorithm.  It is not clear if I
;; need to use this model.
(describe (assert subclass #labAnimals
  superclass( build lex "lab animal")))
(M2!
 (SUBCLASS
```

```
     B1)
 (SUPERCLASS
     (M1
      (LEX lab animal))))

(M2!)

 CPU time : 0.01

*

(describe (build act (build lex "research")
 object *labAnimals) = researchOnLabAnimals)


 CPU time : 0.00

*

;;  The verb is used in what appears to be the transitive sense.
(describe (assert object (build lex "research")
 property (build lex "transitive")))
(M6!
 (OBJECT (M3
          (LEX research)))
 (PROPERTY
     (M5
      (LEX transitive))))

(M6!)

 CPU time : 0.00

*
```

```
;; The moment of truth...

^(

--> defn_verb 'research)

(A SOMETHING CAN RESEARCH A SOMETHING RESULT= NIL ENABLED BY= NIL)





 CPU time : 0.06


*


;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

;;  Representation of:

;;  "stem cells taken from animal embryos"

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;


;; Model the animal embryos used in this experiment.  We are unfamiliar with

;; the type or quantity of of animal embryos used so we consider them to be

;; a subclass of the class animal embryo.

(describe ( assert subclass #animalEmbryosInExperiment

  superclass (build lex "animal embryo")))

(M17!

 (SUBCLASS

     B2)

 (SUPERCLASS

     (M16

      (LEX animal embryo))))


(M17!)


 CPU time : 0.00


*
```

```
;; Specify the subset of stem cells used in this specific research.
;; Assume the class "stem cell" is a basic-level category.  The reason
;; for this is twofold:
;; 1. My first impression was that it was basic-level.
;; 2. Ehrlich's verb algorithm gives better results with basic-level categories
;; as the Agents of actions.
(describe( assert members #stemCellsInExperiment
 class (build lex "stem cell")))
(M19!
 (CLASS (M18
         (LEX stem cell)))
 (MEMBERS B3))

(M19!)

 CPU time : 0.00

*

;; Define that "stem cell" is a basic-level category.
;; This is Ehrlich's model for representing a class as basic-level.
(describe ( assert object1 (lex "stem cell")
  rel ("ISA")
  object2 (build lex "basic ctgy")))
(M21!
 (OBJECT1 LEX
          stem cell)
 (OBJECT2 (M20
           (LEX basic ctgy)))
 (REL ISA))

(M21!)
```

```
 CPU time : 0.00

*

;; Model the action "taken".  The source arc is not part of Ehrlich's case
;; frames.  However, it is not evident that the source of an action is
;; important to the functioning of the verb algorithm.
(describe (build action (build lex "take")
 source *animalEmbryosInExperiment
 object *stemCellsInExperiment) = takeAction)


 CPU time : 0.00

*

;; Show that the set of stem cells is modified by the fact that it is taken
;; from animal embryos.
(describe (build mod takeAction
 head *stemCellsInExperiment) = stemCellsFromResearch)


 CPU time : 0.00

*

;; The moment of truth...
^(
--> defn_verb 'take)
(A SOMETHING CAN TAKE RESULT= NIL ENABLED BY= NIL)
```

```
 CPU time : 0.08

*

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;  Representation of:
;;  "stem cells taken from animal embryos are astoundingly versatile"
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(describe(assert object stemCellsFromResearch
 property (build lex "astoundingly versatile")
= stemCellsAreAstoundinglyVersatile))
(M30!
 (OBJECT STEMCELLSFROMRESEARCH)
 (PROPERTY
     (M29
      (LEX astoundingly versatile))))

(M30!)

 CPU time : 0.00

*

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;  Representation of:
;;  "the research itself revealed that stem cells taken from animal embryos
;;   are astoundingly versatile"
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

;; Ehrlich's algorithm will return the Agent as being SOMETHING rather than making
;; an attempt at guessing what the agent might be.
;; If the user limits himself to using agents that directly or through path based
;; inference are basic ctg, then the algorithm will return useful results.
```

```
;; (Also if the agent is a member of class Animal the algorithm will make a
;; distinction though it is not helpful here.)
;; A possible solution is to model researchOnLabAnimals as a basic-level category.
;; But how?
(describe( assert agent researchOnLabAnimals
  act (build lex "reveal")
  object stemCellsAreAstoundinglyVersatile))
(M32!
 (ACT (M31
       (LEX reveal)))
 (AGENT RESEARCHONLABANIMALS)
 (OBJECT STEMCELLSAREASTOUNDINGLYVERSATILE))

(M32!)

 CPU time : 0.01

*

;; Denote reveal as being a transitive verb in this sense.
(describe (assert object (build lex "reveal")
 property (build lex "transitive")))
(M33!
 (OBJECT (M31
          (LEX reveal)))
 (PROPERTY
     (M5
      (LEX transitive))))

(M33!)

 CPU time : 0.00

*
```

```
;; The moment of truth...
^(
--> defn_verb 'reveal)
(A NIL CAN REVEAL A NIL RESULT= NIL ENABLED BY= NIL)




 CPU time : 0.22


*


;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;  Representation of:
;;  "they grow in the lab"
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

;; The use of grow is modeled as a reflexive verb (the agent acts upon itself).
;; Consider these two sentences:
;; "They grew the watermellon."
;;   Ag   Act       Object
;; and
;; "They grow " (themselves) <- Implied(?)
;;   Ag   Act       Object
;;
;; The location relation is an Ehrlich specific relation.
(describe (assert agent stemCellsFromResearch
  act (build lex "grow")
  object stemCellsFromResearch
  location (build lex "lab") = stemCellsGrownInLab))
(M41!
 (ACT (M39
       (LEX grow)))
 (AGENT STEMCELLSFROMRESEARCH)
```

```
 (LOCATION
     (M40
      (LEX lab)))
 (OBJECT STEMCELLSFROMRESEARCH))

(M41!)

 CPU time : 0.01

*

;; Define grow as being reflexive.  If the Agent associated with grow were
;; a basic ctg or of the class Animal the output would be:
;; "A whatever can grow itself."
(describe(assert object (build lex "grow")
property(build lex "reflexive")))
(M42!
 (OBJECT (M39
          (LEX grow)))
 (PROPERTY
     (M27
      (LEX reflexive))))

(M42!)

 CPU time : 0.00

*

;; The moment of truth...
^(
--> defn_verb 'grow)
(A NIL CAN GROW ITSELF RESULT= NIL ENABLED BY= NIL)
```

```
 CPU time : 0.26

*

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;  Representation of:
;;  "they proliferate like rabbits"
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

;; Show that stem cells proliferate.  Here is a major difference from Marc's
;; original representation.  Marc had the agent representing the conglomerate:
;; "stem cells modified by having been taken from animal embryos."
;; This has two drawbacks:
;; 1.  In the similie "they proliferate like rabbits" they more likely refers
;; to stem cells in general than the specific stem cells obtained from research.
;; This is of course open to debate.
;; 2. Ehrlich's verb algorithm will have difficulty working with an agent that
;; is not a basic ctgy.
;; Therefore I model the agent as being plain old stem cells.
(describe(assert agent  *stemCellsInExperiment
 act (build lex "proliferate")))
(M47!
 (ACT (M46
      (LEX proliferate)))
 (AGENT B3))

(M47!)

 CPU time : 0.00

*
```

```
;; Define the rabbit subset proliferating.  Consider rabbits to be a basic-level category
;; and that stem cells are like a set of these of these rabbits ( a set
;; excluding the Trix bunny, Easter bunny, etc.).
;; This is how Ehrlich represents a collection.
(describe (assert members #rabbits
  class (build lex "rabbit")))
(M49!
 (CLASS (M48
         (LEX rabbit)))
 (MEMBERS B4))

(M49!)

 CPU time : 0.00

*
;; Make rabbits a basic-level category
(describe (assert object1 (build lex "rabbit")
 rel ("ISA")
 object2 (build lex "basic ctgy")))
(M50!
 (OBJECT1 (M48
           (LEX rabbit)))
 (OBJECT2 (M20
           (LEX basic ctgy)))
 (REL ISA))

(M50!)

 CPU time : 0.01

*

;; Show that rabbits proliferate
```

```
(describe(assert agent *rabbits
 act( build lex "proliferate")))
(M51!
 (ACT (M46
       (LEX proliferate)))
 (AGENT B4))

(M51!)

 CPU time : 0.01

*

;; Make proliferate intransitive as it can only be inferred that the verb
;; exits and that it has a subject.
(describe (assert object(build lex "proliferate")
  property(build lex "intransitive")))
(M53!
 (OBJECT (M46
          (LEX proliferate)))
 (PROPERTY
     (M52
      (LEX intransitive))))

(M53!)

 CPU time : 0.01

*

;; The moment of truth...
^(
--> defn_verb 'proliferate)
(A (stem cell
```

```
    rabbit)
   CAN PROLIFERATE RESULT= NIL ENABLED BY= NIL)



 CPU time : 0.52

*

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;; Define a consequence for for the action proliferate.  Ehrlich has a concept
;; of consequence that is very similar to the effect in an action/effect
;; case frame.
;; Model the action proliferate having a consequence as follows:
;; "If r is a rabbit and r proliferates,          <--- Action
;; then r does some unknown consequence."    <--- Consequence of Action
;; "If r is a stem cell and r proliferates, then r does some unknown consequence."
;;
;; It should be noted that consequences are crucial for Ehrlich's algorithm to
;; give a complete verb definition.  Here I represent a consequence that is,
;; at this point, unknown.
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(describe (build lex "unknown consequence")
= unknownCq)


 CPU time : 0.00

*

(describe (assert forall ($rabbits)
  &ant ( (build members *rabbits
class (build lex "rabbit"))
```

```
 (build agent *rabbits
act (build lex "proliferate")))
  cq (build agent *rabbits
    act unknownCq )))
(M58!
 (FORALL V47)
 (&ANT (P77
        (ACT (M46
              (LEX proliferate)))
        (AGENT V47))
       (P76
        (CLASS (M48
                (LEX rabbit)))
        (MEMBERS V47)))
 (CQ (P78
      (ACT UNKNOWNCQ)
      (AGENT V47))))

(M58!)

 CPU time : 0.00

*



;; Proof that the rule is in working order
(describe (deduce agent $x act unknownCq ))
(M59!
 (ACT UNKNOWNCQ)
 (AGENT B4))

(M59!)

 CPU time : 0.09
```

```
*

(describe (assert forall ($sc)
  &ant ( (build members *sc
class (build lex "stem cell"))
 (build agent *sc
act (build lex "proliferate"))
)
  cq (build agent *sc
    act unknownCq )))
(M60!
 (FORALL V49)
 (&ANT (P81
        (ACT (M46
              (LEX proliferate)))
        (AGENT V49))
       (P80
        (CLASS (M18
                (LEX stem cell)))
        (MEMBERS V49)))
 (CQ (P82
      (ACT UNKNOWNCQ)
      (AGENT V49))))

(M60!)

 CPU time : 0.01

*

;; Proof that the rule is in working order
```

```
(describe (deduce agent $x act unknownCq ))
(M61!
 (ACT UNKNOWNCQ)
 (AGENT B3))
(M59!
 (ACT UNKNOWNCQ)
 (AGENT B4))

(M61!
 M59!)

 CPU time : 0.09

*

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;; Make an attempt at defining the similie/metaphor relation.
;; Essentially, the unknown consequence for a rabbit proliferating is in a
;; metaphor/similie action relation to the unknown consequence for a stem cell
;; proliferating on some conceptual level.
;; Similar to Marc's representation if object1 is viewed as source, object2
;; is viewed as target and rel defines the relation as being a "similie/metaphor"
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(describe (assert object1 unknownCq
  rel (build mod ("unknown concept")
       head ("similie/metaphor action"))
  object2 unknownCq ))
(M63!
 (OBJECT1 UNKNOWNCQ)
 (OBJECT2 UNKNOWNCQ)
 (REL (M62
       (HEAD similie/metaphor action)
       (MOD unknown concept))))
```

```
(M63!)

 CPU time : 0.01

*

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;; object1/rel/object2 can have further meaning if we create a specific rule
;; with which to interpret the case frame when rel points to a "similie/metaphor
;; action".  For instance:
;; "If
;; conseqeunce x and consequnece y are in a "simile/metaphor action" relation
;;  modified by context z,
;; then
;; similarities exist between consequence x and consequence y based on the
;; context z."
;;
;; For instance if the sentence were "Juliet runs like a deer." then there are
;; two consequences for running (or less, depending on how it is modeled.)  The
;; consequences could be:
;; 1. If Juliet runs she moves from point A to point B quickly.
;; 2. If a deer runs it moves from point A to point B quickly.
;; Both of these consequences would be similar based on the concept of
;; "gracefullness."
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(describe ( assert forall ($x $y $z)
        &ant (build object1 *x
       rel (build mod *z
                         head ("similie/metaphor action"))
             object2 *y  )
        cq (build head (build agent (build lex "similarity(s)")
             act (build lex "exist between")
```

```
           object *y
             object *x)
             mod *z)))
(M66!
 (FORALL V53
         V52
         V51)
 (&ANT (P85
        (OBJECT1 V51)
        (OBJECT2 V52)
        (REL (P84
              (HEAD similie/metaphor action)
              (MOD V53)))))
 (CQ (P87
      (HEAD (P86
             (ACT (M65
                   (LEX exist between)))
             (AGENT (M64
                     (LEX similarity(s))))
             (OBJECT V52
                     V51)))
      (MOD V53))))

(M66!)

 CPU time : 0.00

*

;; Proof that the rule is in working order
(describe (deduce head (build act (build lex "exist between"))))



 CPU time : 0.07
```

```
*

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;; "They turn into specialized cells such as neurons."
;; Define the neurons and specialized cells
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

;; Make neurons basic-level
(describe ( assert object1 (build lex "neuron")
  rel ("ISA")
  object2 (build lex "basic ctgy")))
(M70!
 (OBJECT1 (M69
           (LEX neuron)))
 (OBJECT2 (M20
           (LEX basic ctgy)))
 (REL ISA))

(M70!)

 CPU time : 0.00

*

;; Make neurons a subclass of specialized cells
(describe (assert subclass (build lex "neuron")
  superclass (build lex "specialized cell")))
(M72!
 (SUBCLASS
     (M69
      (LEX neuron)))
 (SUPERCLASS
     (M71
```

```
      (LEX specialized cell))))

(M72!)

 CPU time : 0.00

*

;; Model the neurons in research as a collection of the class neuron.
;; Uses an Ehrlich specific case frame.
(describe (assert members #neurons
  class (build lex "neuron")))
(M73!
 (CLASS (M69
         (LEX neuron)))
 (MEMBERS B5))

(M73!)

 CPU time : 0.00

*

;; Model the action of turning into something.
(describe (assert agent stemCellsFromResearch
  act (build lex "turn into")
  object *neurons))
(M75!
 (ACT (M74
       (LEX turn into)))
 (AGENT STEMCELLSFROMRESEARCH)
 (OBJECT B5))

(M75!)
```

```
 CPU time : 0.00

*

(describe (build object(build lex "turn into")
 property (build lex "transitive")))


 CPU time : 0.00

*

;; The moment of truth...
^(
--> defn_verb 'proliferate)
(A (stem cell
    rabbit)
   CAN PROLIFERATE RESULT=
   (P82
    P78)
   ENABLED BY= NIL)

 CPU time : 0.93

*
End of #p/doc/graduate/663/demos/proliferate.demo demonstration.


 CPU time : 2.75

*
```

Appendix D

Source code for the "proliferate" passage.

```
;; A representation of:
;; "A decade ago research on lab animals revealed that stem cells taken from
;;  animal embryos are astoundingly versatile.  They grow in the lab, proliferate
;;  like rabbits and turn into specialized cells such as neurons (Begley 2001)."
;;
;; Representations are based on Marc Broklawski's original representations though
;; changes have been made to adhere to the case frames used by Ehrlich's
;; algorithm.  Also new rules have been introduced.


;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;  Setup work.  Define external paths and relations and load verb algorithm
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;


;;; Reset the network
(resetnet t)


;; Define the relations we will need.
(define agent act object lext property objects1 object1 rel object2 ant cq &cq members class mod head source target transference vehicle forall &ant context arg1 arg2)


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


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


;;; Compose paths
(intext "paths")


;;; Load Noun/Verb Algorithm
^(load "code")
```

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;  Representation of:
;;  "research on lab animals"
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

;; Model the subclass of lab animals upon which research was performed.  A
;; subclass/superclass case frmae is used.
;; Ehrlich might not have used a subclass/superclass to represent this set.  She
;; instead might have done the following:
;;
;; object1 #setOfLabAnimals
;; rel "ARE"
;; object2  (build lex "lab animal")
;;
;; a variant which she admits is far from ideal but might be
;; needed for the purposes of her algorithm.  It is not clear if I
;; need to use this model.
(describe (assert subclass #labAnimals
  superclass( build lex "lab animal")))

(describe (build act (build lex "research")
 object *labAnimals) = researchOnLabAnimals)

;;  The verb is used in what appears to be the transitive sense.
(describe (assert object (build lex "research")
 property (build lex "transitive")))

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

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;  Representation of:
;;  "stem cells taken from animal embryos"
```

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;


;; Model the animal embryos used in this experiment.  We are unfamiliar with
;; the type or quantity of of animal embryos used so we consider them to be
;; a subclass of the class animal embryo.
(describe ( assert subclass #animalEmbryosInExperiment
  superclass (build lex "animal embryo")))



;; Specify the subset of stem cells used in this specific research.
;; Assume the class "stem cell" is a basic-level category.  The reason
;; for this is twofold:
;; 1. My first impression was that it was basic-level.
;; 2. Ehrlich's verb algorithm gives better results with basic-level categories
;; as the Agents of actions.
(describe( assert members #stemCellsInExperiment
 class (build lex "stem cell")))


;; Define that "stem cell" is a basic-level category.
;; This is Ehrlich's model for representing a class as basic-level.
(describe ( assert object1 (lex "stem cell")
  rel ("ISA")
  object2 (build lex "basic ctgy")))


;; Model the action "taken".  The source arc is not part of Ehrlich's case
;; frames.  However, it is not evident that the source of an action is
;; important to the functioning of the verb algorithm.
(describe (build action (build lex "take")
 source *animalEmbryosInExperiment
 object *stemCellsInExperiment) = takeAction)


;; Show that the set of stem cells is modified by the fact that it is taken
;; from animal embryos.
(describe (build mod takeAction
```

```
 head *stemCellsInExperiment) = stemCellsFromResearch)

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

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;  Representation of:
;;  "stem cells taken from animal embryos are astoundingly versatile"
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(describe(assert object stemCellsFromResearch
 property (build lex "astoundingly versatile")
= stemCellsAreAstoundinglyVersatile))

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;  Representation of:
;;  "the research itself revealed that stem cells taken from animal embryos
;;   are astoundingly versatile"
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

;; Ehrlich's algorithm will return the Agent as being SOMETHING rather than making
;; an attempt at guessing what the agent might be.
;; If the user limits himself to using agents that directly or through path based
;; inference are basic ctg, then the algorithm will return useful results.
;; (Also if the agent is a member of class Animal the algorithm will make a
;; distinction though it is not helpful here.)
;; A possible solution is to model researchOnLabAnimals as a basic-level category.
;; But how?
(describe( assert agent researchOnLabAnimals
  act (build lex "reveal")
  object stemCellsAreAstoundinglyVersatile))

;; Denote reveal as being a transitive verb in this sense.
(describe (assert object (build lex "reveal")
```

```
 property (build lex "transitive")))

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

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;  Representation of:
;;  "they grow in the lab"
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

;; The use of grow is modeled as a reflexive verb (the agent acts upon itself).
;; Consider these two sentences:
;; "They grew the watermellon."
;;   Ag   Act       Object
;; and
;; "They grow " (themselves) <- Implied(?)
;;   Ag   Act       Object
;;
;; The location relation is an Ehrlich specific relation.
(describe (assert agent stemCellsFromResearch
  act (build lex "grow")
  object stemCellsFromResearch
  location (build lex "lab") = stemCellsGrownInLab))

;; Define grow as being reflexive.  If the Agent associated with grow were
;; a basic ctg or of the class Animal the output would be:
;; "A whatever can grow itself."
(describe(assert object (build lex "grow")
property(build lex "reflexive")))

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

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
```

```
;;  Representation of:
;;  "they proliferate like rabbits"
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

;; Show that stem cells proliferate.  Here is a major difference from Marc's
;; original representation.  Marc had the agent representing the conglomerate:
;; "stem cells modified by having been taken from animal embryos."
;; This has two drawbacks:
;; 1.  In the similie "they proliferate like rabbits" they more likely refers
;; to stem cells in general than the specific stem cells obtained from research.
;; This is of course open to debate.
;; 2. Ehrlich's verb algorithm will have difficulty working with an agent that
;; is not a basic ctgy.
;; Therefore I model the agent as being plain old stem cells.
(describe(assert agent  *stemCellsInExperiment
 act (build lex "proliferate")))

;; Define the rabbit subset proliferating.  Consider rabbits to be a basic-level category
;; and that stem cells are like a set of these of these rabbits ( a set
;; excluding the Trix bunny, Easter bunny, etc.).
;; This is how Ehrlich represents a collection.
(describe (assert members #rabbits
  class (build lex "rabbit")))
;; Make rabbits a basic-level category
(describe (assert object1 (build lex "rabbit")
 rel ("ISA")
 object2 (build lex "basic ctgy")))

;; Show that rabbits proliferate
(describe(assert agent *rabbits
 act( build lex "proliferate")))

;; Make proliferate intransitive as it can only be inferred that the verb
;; exits and that it has a subject.
```

```
(describe (assert object(build lex "proliferate")
  property(build lex "intransitive")))

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

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;; Define a consequence for for the action proliferate.  Ehrlich has a concept
;; of consequence that is very similar to the effect in an action/effect
;; case frame.
;; Model the action proliferate having a consequence as follows:
;; "If r is a rabbit and r proliferates,          <--- Action
;; then r does some unknown consequence."    <--- Consequence of Action
;; "If r is a stem cell and r proliferates, then r does some unknown consequence."
;;
;; It should be noted that consequences are crucial for Ehrlich's algorithm to
;; give a complete verb definition.  Here I represent a consequence that is,
;; at this point, unknown.
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(describe (build lex "unknown consequence")
= unknownCq)

(describe (assert forall ($rabbits)
  &ant ( (build members *rabbits
class (build lex "rabbit"))
 (build agent *rabbits
act (build lex "proliferate")))
  cq (build agent *rabbits
    act unknownCq )))



;; Proof that the rule is in working order
(describe (deduce agent $x act unknownCq ))
```

```
(describe (assert forall ($sc)
  &ant ( (build members *sc
class (build lex "stem cell"))
 (build agent *sc
act (build lex "proliferate"))
)
  cq (build agent *sc
    act unknownCq )))


;; Proof that the rule is in working order
(describe (deduce agent $x act unknownCq ))

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;; Make an attempt at defining the similie/metaphor relation.
;; Essentially, the unknown consequence for a rabbit proliferating is in a
;; metaphor/similie action relation to the unknown consequence for a stem cell
;; proliferating on some conceptual level.
;; Similar to Marc's representation if object1 is viewed as source, object2
;; is viewed as target and rel defines the relation as being a "similie/metaphor"
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(describe (assert object1 unknownCq
  rel (build mod ("unknown concept")
       head ("similie/metaphor action"))
  object2 unknownCq ))

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;; object1/rel/object2 can have further meaning if we create a specific rule
;; with which to interpret the case frame when rel points to a "similie/metaphor
;; action".  For instance:
;; "If
```

```
;; conseqeunce x and consequnece y are in a "simile/metaphor action" relation
;;  modified by context z,
;; then
;; similarities exist between consequence x and consequence y based on the
;; context z."
;;
;; For instance if the sentence were "Juliet runs like a deer." then there are
;; two consequences for running (or less, depending on how it is modeled.)  The
;; consequences could be:
;; 1. If Juliet runs she moves from point A to point B quickly.
;; 2. If a deer runs it moves from point A to point B quickly.
;; Both of these consequences would be similar based on the concept of
;; "gracefullness."
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(describe ( assert forall ($x $y $z)
        &ant (build object1 *x
       rel (build mod *z
                  head ("similie/metaphor action"))
             object2 *y  )
        cq (build head (build agent (build lex "similarity(s)")
             act (build lex "exist between")
           object *y
             object *x)
             mod *z)))

;; Proof that the rule is in working order
(describe (deduce head (build act (build lex "exist between"))))

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;; "They turn into specialized cells such as neurons."
;; Define the neurons and specialized cells
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
```

```
;; Make neurons basic-level
(describe ( assert object1 (build lex "neuron")
  rel ("ISA")
  object2 (build lex "basic ctgy")))

;; Make neurons a subclass of specialized cells
(describe (assert subclass (build lex "neuron")
  superclass (build lex "specialized cell")))

;; Model the neurons in research as a collection of the class neuron.
;; Uses an Ehrlich specific case frame.
(describe (assert members #neurons
  class (build lex "neuron")))

;; Model the action of turning into something.
(describe (assert agent stemCellsFromResearch
  act (build lex "turn into")
  object *neurons))

(describe (build object(build lex "turn into")
 property (build lex "transitive")))

;; The moment of truth...
^(defn_verb 'proliferate)
```

Appendix E

SNePS output for the "proliferate" background knowledge.

```
* (demo "probk.demo")



File #p/doc/graduate/663/demos/probk.demo is now the source of input.



 CPU time : 0.00


* ;;; Link to the representation of the passage.

(intext "proliferate.demo")

File proliferate.demo is now the source of input.



 CPU time : 0.00


*


Net reset



 CPU time : 0.01


* ACT is already defined.

OBJECT1 is already defined.

OBJECT2 is already defined.

ANT is already defined.

CQ is already defined.

FORALL is already defined.

&ANT is already defined.


(AGENT ACT OBJECT LEXT PROPERTY OBJECTS1 OBJECT1 REL OBJECT2 ANT CQ &CQ
```

```
       MEMBERS CLASS MOD HEAD SOURCE TARGET TRANSFERENCE VEHICLE FORALL
       &ANT CONTEXT ARG1 ARG2)

 CPU time : 0.00

*
-->
NIL

 CPU time : 0.00

*
File rels is now the source of input.

 CPU time : 0.00

* ACT is already defined.
ACTION is already defined.
AGENT is already defined.
CLASS is already defined.
EFFECT is already defined.
MEMBERS is already defined.
OBJECT is already defined.
OBJECT1 is already defined.
OBJECTS1 is already defined.
OBJECT2 is already defined.
PROPERTY is already defined.
REL 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
    POSSESOR)

 CPU time : 0.04

*

End of file rels

 CPU time : 0.04

*
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 : 0.00

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

```
 CPU time : 0.00

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

 CPU time : 0.00

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

 CPU time : 0.00

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

 CPU time : 0.00

*

End of file paths
```

```
 CPU time : 0.01


*

-->

Loading code

Finished loading code

T




 CPU time : 0.18


*

(M2!

 (SUBCLASS

     B1)

 (SUPERCLASS

     (M1

      (LEX lab animal))))


(M2!)


 CPU time : 0.00


*



 CPU time : 0.00


*

(M6!

 (OBJECT (M3

          (LEX research)))

 (PROPERTY
```

```
     (M5
      (LEX transitive))))

(M6!)

 CPU time : 0.00

*
-->
(A SOMETHING CAN RESEARCH A SOMETHING RESULT= NIL ENABLED BY= NIL)



 CPU time : 0.03

*
(M17!
 (SUBCLASS
     B2)
 (SUPERCLASS
     (M16
      (LEX animal embryo))))

(M17!)

 CPU time : 0.00

*
(M19!
 (CLASS (M18
         (LEX stem cell)))
 (MEMBERS B3))

(M19!)
```

```
 CPU time : 0.00


*

(M21!

 (OBJECT1 LEX

          stem cell)

 (OBJECT2 (M20

           (LEX basic ctgy)))

 (REL ISA))


(M21!)


 CPU time : 0.00


*



 CPU time : 0.00


*



 CPU time : 0.01


*

-->

(A SOMETHING CAN TAKE RESULT= NIL ENABLED BY= NIL)




 CPU time : 0.09


*
```

```
(M30!
 (OBJECT STEMCELLSFROMRESEARCH)
 (PROPERTY
     (M29
      (LEX astoundingly versatile))))

(M30!)

 CPU time : 0.00

*
(M32!
 (ACT (M31
       (LEX reveal)))
 (AGENT RESEARCHONLABANIMALS)
 (OBJECT STEMCELLSAREASTOUNDINGLYVERSATILE))

(M32!)

 CPU time : 0.00

*
(M33!
 (OBJECT (M31
          (LEX reveal)))
 (PROPERTY
     (M5
      (LEX transitive))))

(M33!)

 CPU time : 0.00

*
```

```
-->

(A NIL CAN REVEAL A NIL RESULT= NIL ENABLED BY= NIL)




 CPU time : 0.31


*

(M41!

 (ACT (M39

       (LEX grow)))

 (AGENT STEMCELLSFROMRESEARCH)

 (LOCATION

     (M40

      (LEX lab)))

 (OBJECT STEMCELLSFROMRESEARCH))


(M41!)


 CPU time : 0.01


*

(M42!

 (OBJECT (M39

          (LEX grow)))

 (PROPERTY

     (M27

      (LEX reflexive))))


(M42!)


 CPU time : 0.00


*
```

```
-->
(A NIL CAN GROW ITSELF RESULT= NIL ENABLED BY= NIL)




 CPU time : 0.27


*
(M47!
 (ACT (M46
       (LEX proliferate)))
 (AGENT B3))

(M47!)


 CPU time : 0.00


*
(M49!
 (CLASS (M48
         (LEX rabbit)))
 (MEMBERS B4))

(M49!)


 CPU time : 0.00


*
(M50!
 (OBJECT1 (M48
           (LEX rabbit)))
 (OBJECT2 (M20
           (LEX basic ctgy)))
 (REL ISA))
```

```
(M50!)

 CPU time : 0.00

*
(M51!
 (ACT (M46
       (LEX proliferate)))
 (AGENT B4))

(M51!)

 CPU time : 0.01

*
(M53!
 (OBJECT (M46
          (LEX proliferate)))
 (PROPERTY
     (M52
      (LEX intransitive))))

(M53!)

 CPU time : 0.00

*
-->
(A (stem cell
    rabbit)
   CAN PROLIFERATE RESULT= NIL ENABLED BY= NIL)
```

```
 CPU time : 0.57

*



 CPU time : 0.00


*
(M58!
 (FORALL V47)
 (&ANT (P77
        (ACT (M46
              (LEX proliferate)))
        (AGENT V47))
       (P76
        (CLASS (M48
                (LEX rabbit)))
        (MEMBERS V47)))
 (CQ (P78
      (ACT UNKNOWNCQ)
      (AGENT V47))))

(M58!)


 CPU time : 0.01


*
(M59!
 (ACT UNKNOWNCQ)
 (AGENT B4))

(M59!)
```

```
 CPU time : 0.05

*
(M60!
 (FORALL V49)
 (&ANT (P81
        (ACT (M46
              (LEX proliferate)))
        (AGENT V49))
       (P80
        (CLASS (M18
                (LEX stem cell)))
        (MEMBERS V49)))
 (CQ (P82
      (ACT UNKNOWNCQ)
      (AGENT V49))))

(M60!)

 CPU time : 0.01

*
(M61!
 (ACT UNKNOWNCQ)
 (AGENT B3))
(M59!
 (ACT UNKNOWNCQ)
 (AGENT B4))

(M61!
 M59!)

 CPU time : 0.14
```

```
*
(M63!
 (OBJECT1 UNKNOWNCQ)
 (OBJECT2 UNKNOWNCQ)
 (REL (M62
       (HEAD similie/metaphor action)
       (MOD unknown concept))))

(M63!)

 CPU time : 0.01

*
(M66!
 (FORALL V53
         V52
         V51)
 (&ANT (P85
        (OBJECT1 V51)
        (OBJECT2 V52)
        (REL (P84
              (HEAD similie/metaphor action)
              (MOD V53)))))
 (CQ (P87
      (HEAD (P86
             (ACT (M65
                   (LEX exist between)))
             (AGENT (M64
                     (LEX similarity(s))))
             (OBJECT V52
                     V51)))
      (MOD V53))))

(M66!)
```

```
 CPU time : 0.01

*



 CPU time : 0.03


*
(M70!
 (OBJECT1 (M69
           (LEX neuron)))
 (OBJECT2 (M20
           (LEX basic ctgy)))
 (REL ISA))

(M70!)

 CPU time : 0.00


*
(M72!
 (SUBCLASS
     (M69
      (LEX neuron)))
 (SUPERCLASS
     (M71
      (LEX specialized cell))))

(M72!)

 CPU time : 0.00


*
```

```
(M73!
 (CLASS (M69
         (LEX neuron)))
 (MEMBERS B5))

(M73!)

 CPU time : 0.00

*
(M75!
 (ACT (M74
       (LEX turn into)))
 (AGENT STEMCELLSFROMRESEARCH)
 (OBJECT B5))

(M75!)

 CPU time : 0.00

*


 CPU time : 0.00

*
-->
(A (stem cell
    rabbit)
   CAN PROLIFERATE RESULT=
   (P82
    P78)
   ENABLED BY= NIL)
```

```
 CPU time : 1.04

*

End of file proliferate.demo

 CPU time : 2.87

*

^(
--> defn_verb 'proliferate)
(A (stem cell
    rabbit)
   CAN PROLIFERATE RESULT=
   (P82
    P78)
   ENABLED BY= NIL)

 CPU time : 1.08

*

;; Define a taxonomy that includes the class for rabbits, stem cells and several parent
;; classes.  The parent class common to both is "animal"

(describe (assert subclass (build lex "rabbit")
  superclass (build lex "mammal")))
```

```
(M78!
 (SUBCLASS
     (M48
      (LEX rabbit)))
 (SUPERCLASS
     (M77
      (LEX mammal))))

(M78!)

 CPU time : 0.00

*

(describe (assert subclass (build lex "mammal")
  superclass (build lex "animal")))
(M79!
 (SUBCLASS
     (M77
      (LEX mammal)))
 (SUPERCLASS
     (M38
      (LEX animal))))

(M79!)

 CPU time : 0.00

*

(describe (assert subclass (build lex "cell")
  superclass (build lex "animal")))
(M81!
 (SUBCLASS
```

```
     (M80
      (LEX cell)))
 (SUPERCLASS
     (M38
      (LEX animal))))

(M81!)

 CPU time : 0.01

*

(describe (assert subclass (build lex "stem cell")
  superclass (build lex "cell")))
(M82!
 (SUBCLASS
     (M18
      (LEX stem cell)))
 (SUPERCLASS
     (M80
      (LEX cell))))

(M82!)

 CPU time : 0.00

*

;; Needed for path based inference.
(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


*



;; Based on the above superclass/subclass scheme ( a watered down taxonomy) implement
;; specific characteristics for classes.


(describe (assert object (build lex "rabbit")
  property (build lex "cute")))
(M84!
 (OBJECT (M48
          (LEX rabbit)))
 (PROPERTY
     (M83
      (LEX cute))))


(M84!)


 CPU time : 0.00


*


(describe (assert object (build lex "rabbit")
  property (build lex "quick")))
(M86!
 (OBJECT (M48
```

```
          (LEX rabbit)))
 (PROPERTY
     (M85
      (LEX quick))))

(M86!)

 CPU time : 0.00

*

(describe (assert object (build lex "rabbit")
  property( build lex "quick birth rate")))
(M88!
 (OBJECT (M48
          (LEX rabbit)))
 (PROPERTY
     (M87
      (LEX quick birth rate))))

(M88!)

 CPU time : 0.00

*

(describe (assert object (build lex "rabbit")
  property (build lex "hairy")))
(M90!
 (OBJECT (M48
          (LEX rabbit)))
 (PROPERTY
     (M89
      (LEX hairy))))
```

```
(M90!)

 CPU time : 0.00

*

(describe (assert object (build lex "rabbit")
  property (build lex "warm blooded")))
(M92!
 (OBJECT (M48
          (LEX rabbit)))
 (PROPERTY
     (M91
      (LEX warm blooded))))

(M92!)

 CPU time : 0.00

*

;; It was difficult to model specific characteristics of cells.  At the least
;; things that stem cells possess can be modeled.

(describe (assert possessor (build lex "stem cell")
  rel (build lex "anatomical element")
  object (build lex "cell wall")))
(M95!
 (OBJECT (M94
          (LEX cell wall)))
 (POSSESSOR
     (M18
      (LEX stem cell)))
```

```
 (REL (M93
       (LEX anatomical element))))

(M95!)

 CPU time : 0.00

*

(describe (assert possesor (build lex "stem cell")
  rel (build lex "anatomical element")
  object  (build lex "nucleus")))
(M97!
 (OBJECT (M96
          (LEX nucleus)))
 (POSSESOR
     (M18
      (LEX stem cell)))
 (REL (M93
       (LEX anatomical element))))

(M97!)

 CPU time : 0.00

*

(describe (assert possessor (build lex "stem cell")
  rel (build lex "anatomical element")
  object  (build lex "protoplasm")))
(M99!
 (OBJECT (M98
          (LEX protoplasm)))
 (POSSESSOR
```

```
     (M18
      (LEX stem cell)))
 (REL (M93
       (LEX anatomical element))))

(M99!)

 CPU time : 0.01

*

;; Based on the above superclass/subclass scheme ( a watered down taxonomy) implement
;; specific actions for specific classes.  An important distinction can be made here
;; and it can be shown that though sufficient for this example, mapping actions to classes
;; represented in a taxonomical hierarchy may not always be advisable.  Discussed in report.
;;
;; Consider the action walk.  Walking crosses many different classes (lots of mammals
;; walk, lots of reptiles walk, lots of birds walk, etc.) but it is not the case
;; that all mammals walk (dolphins and whales) or all reptiles walk (snakes).  A
;; hierarchy not based on taxonomy would be needed, for instance legged animals as
;; a superclass with four legged and two legged as descending subclasses.  Actions
;; would then be mappped to the calss four legged animals, of which rabbits would
;; be either a member or possibly a subclass.
;;
;; Because stem cells and rabbits are so different (in the sense of actions both can perform
;; like living, dying, reproducing, etc.) a taxonomical hierarchy is sufficient for
;; categorizing actions.
;;

;; Categorize an action for rabbits

(describe (assert forall ($r $rs)
```

```
  ant ( (build member *r
       class (build lex "rabbit"))
           (build members *rs
       class (build lex "rabbit")))
  cq ( (build agent *r
      act (build lex "hop"))
  (build agent *rs
      act (build lex "hop")))
;; could be any number of rabbit specific actions here
))
(M101!
 (FORALL V75
         V74)
 (ANT (P121
       (CLASS (M48
               (LEX rabbit)))
       (MEMBERS V75))
      (P120
       (CLASS (M48))
       (MEMBER V74)))
 (CQ (P123
      (ACT (M100
            (LEX hop)))
      (AGENT V75))
     (P122
      (ACT (M100))
      (AGENT V74))))

(M101!)

 CPU time : 0.05

*
```

```
;; Proof that the rule is in working order

(describe (deduce agent $x act (build lex "hop")))

(M102!

 (ACT (M100

       (LEX hop)))

 (AGENT B4))

(P122!

 (ACT (M100))

 (AGENT V74))


(M102!

 P122!)


 CPU time : 0.21


*


;; Categorize the actions for cells


(describe (assert forall ($c $cs)

  ant ( (build member *c

       class (build lex "cell"))

           (build members *cs

       class (build lex "cell")))

  cq ( (build agent *c

      act (build lex "divide"))

  (build agent *cs

      act (build lex "divide")))

;; could be any number of cell specific actions here

))

(M104!

 (FORALL V78

         V77)

 (ANT (P126
```

```
       (CLASS (M80
               (LEX cell)))
       (MEMBERS V78))
      (P125
       (CLASS (M80))
       (MEMBER V77)))
 (CQ (P128
      (ACT (M103
            (LEX divide)))
      (AGENT V78))
     (P127
      (ACT (M103))
      (AGENT V77))))

(M104!)

 CPU time : 0.00

*

;; Proof that the rule is in working order
(describe (deduce agent $x act (build lex "divide")))
(M106!
 (ACT (M103
       (LEX divide)))
 (AGENT B3))
(P127!
 (ACT (M103))
 (AGENT V77))

(M106!
 P127!)

 CPU time : 0.27
```

*

```
;; Categorize actions common to both stem cells and rabbits.  This probably
;; ought to be realized by heirarchy structure.

(describe (assert forall ($a $as $r $rs)
  ant ( (build member *a
       class (build lex "stem cell"))
           (build members *as
       class (build lex "stem cell"))
 (build member *a
       class (build lex "rabbit"))
           (build members *as
       class (build lex "rabbit")))
  cq ( (build agent *a
      act (build lex "live"))
  (build agent *as
      act (build lex "live"))
(build agent *a
      act (build lex "die"))
  (build agent *as
      act (build lex "die"))
(build agent *a
      act (build lex "reproduce"))
  (build agent *as
      act (build lex "reproduce"))
  (build agent *r
      act (build lex "live"))
  (build agent *rs
      act (build lex "live"))
(build agent *r
      act (build lex "die"))
  (build agent *rs
```

```
      act (build lex "die"))
(build agent *r
      act (build lex "reproduce"))
  (build agent *rs
      act (build lex "reproduce")))
;; could be any number of cell specific actions here
))
(M110!
 (FORALL V83
         V82
         V81
         V80)
 (ANT (P133
       (CLASS (M48
               (LEX rabbit)))
       (MEMBERS V81))
      (P132
       (CLASS (M48))
       (MEMBER V80))
      (P131
       (CLASS (M18
               (LEX stem cell)))
       (MEMBERS V81))
      (P130
       (CLASS (M18))
       (MEMBER V80)))
 (CQ (P145
      (ACT (M109
            (LEX reproduce)))
      (AGENT V83))
     (P144
      (ACT (M109))
      (AGENT V82))
     (P143
```

```
 (ACT (M108
       (LEX die)))
 (AGENT V83))
(P142
 (ACT (M108))
 (AGENT V82))
(P141
 (ACT (M107
       (LEX live)))
 (AGENT V83))
(P140
 (ACT (M107))
 (AGENT V82))
(P139
 (ACT (M109))
 (AGENT V81))
(P138
 (ACT (M109))
 (AGENT V80))
(P137
 (ACT (M108))
 (AGENT V81))
(P136
 (ACT (M108))
 (AGENT V80))
(P135
 (ACT (M107))
 (AGENT V81))
(P134
 (ACT (M107))
 (AGENT V80))))

(M110!)
```

```
 CPU time : 0.06

*

;; Proof that the rule is in working order
(describe (deduce agent $x act (build lex "live")))
(M112!
 (ACT (M107
       (LEX live)))
 (AGENT B3))
(M111!
 (ACT (M107))
 (AGENT B4))
(P141!
 (ACT (M107))
 (AGENT V83))
(P140!
 (ACT (M107))
 (AGENT V82))
(P134!
 (ACT (M107))
 (AGENT V80))

(M112!
 M111!
 P141!
 P140!
 P134!)

 CPU time : 0.57

*

;; There now exists five actions.  For each of these actions define consequences.
```

```
;; A consequence for hopping

(describe (assert forall $e

  ant((build agent *e

     act (build lex "hop")

))

  cq((build agent *e

    act(build mod (build lex "jumping")

      head (build lex "move"))))

))

(M116!

 (FORALL V85)

 (ANT (P147

       (ACT (M100

             (LEX hop)))

       (AGENT V85)))

 (CQ (P148

      (ACT (M115

            (HEAD (M114

                   (LEX move)))

            (MOD (M113

                  (LEX jumping)))))

      (AGENT V85))))


(M116!)


 CPU time : 0.01


*


;; Proof that the rule is in working order

(describe (deduce agent $x act(build mod (build lex "jumping")

     head (build lex "move"))))

(M117!
```

```
 (ACT (M115
       (HEAD (M114
              (LEX move)))
       (MOD (M113
             (LEX jumping)))))
 (AGENT B4))
(P150!
 (ACT (M115))
 (AGENT V74))

(M117!
 P150!)

 CPU time : 0.19

*

;; A consequence for dividing
(describe (assert forall $e
  ant((build agent *e
     act (build lex "divide")
))
  cq((build agent *e
    act(build mod (build lex "evenly")
      head (build lex "splits"))))
))
(M121!
 (FORALL V87)
 (ANT (P151
       (ACT (M103
             (LEX divide)))
       (AGENT V87)))
 (CQ (P152
      (ACT (M120
```

```
            (HEAD (M119
                   (LEX splits)))
            (MOD (M118
                  (LEX evenly)))))
      (AGENT V87))))

(M121!)

 CPU time : 0.00

*

;; Proof that the rule is in working order
(describe (deduce agent $x act(build mod (build lex "evenly")
     head (build lex "splits"))))
(M122!
 (ACT (M120
       (HEAD (M119
              (LEX splits)))
       (MOD (M118
             (LEX evenly)))))
 (AGENT B3))
(P154!
 (ACT (M120))
 (AGENT V77))

(M122!
 P154!)

 CPU time : 0.21

*

;; A consequence for living
```

```
(describe (assert forall $e
  ant((build agent *e
     act (build lex "live")
))
  cq((build agent *e
    act (build lex "perform")
    object (build lex "actions")))
))
(M125!
 (FORALL V89)
 (ANT (P155
       (ACT (M107
             (LEX live)))
       (AGENT V89)))
 (CQ (P156
      (ACT (M123
            (LEX perform)))
      (AGENT V89)
      (OBJECT (M124
               (LEX actions))))))

(M125!)

 CPU time : 0.01

*

;; Proof that the rule is in working order
(describe (deduce agent $x act(build lex "perform")))
(P163!
 (ACT (M123
       (LEX perform)))
 (AGENT V83))
(P161!
```

```
 (ACT (M123))
 (AGENT V82))
(M129!
 (ACT (M123))
 (AGENT B4))
(M128!
 (ACT (M123))
 (AGENT B3))
(P159!
 (ACT (M123))
 (AGENT V80))

(P163!
 P161!
 M129!
 M128!
 P159!)

 CPU time : 0.25

*

;; A consequence for dying
(describe (assert forall $e
  ant((build agent *e
     act (build lex "die")
))
  cq((build agent *e
    act (build lex "stop")
    object (build lex "living")))
))
(M132!
 (FORALL V91)
```

```
 (ANT (P164
       (ACT (M108
             (LEX die)))
       (AGENT V91)))
 (CQ (P165
      (ACT (M130
            (LEX stop)))
      (AGENT V91)
      (OBJECT (M131
               (LEX living))))))

(M132!)

 CPU time : 0.00

*

;; Proof that the rule is in working order
(describe (deduce agent $x act(build lex "stop")))
(M138!
 (ACT (M130
       (LEX stop)))
 (AGENT B3))
(P172!
 (ACT (M130))
 (AGENT V80))
(M135!
 (ACT (M130))
 (AGENT B4))
(P171!
 (ACT (M130))
 (AGENT V82))
(P170!
 (ACT (M130))
```

```
 (AGENT V83))

(M138!
 P172!
 M135!
 P171!
 P170!)

 CPU time : 0.13

*

;; A consequence for reproducing
(describe (assert forall $e
  ant((build agent *e
     act (build lex "reproduce")
))
  cq((build agent *e
    act (build lex "produce")
    object (build lex "offspring")))
))
(M141!
 (FORALL V93)
 (ANT (P173
       (ACT (M109
             (LEX reproduce)))
       (AGENT V93)))
 (CQ (P174
      (ACT (M139
            (LEX produce)))
      (AGENT V93)
      (OBJECT (M140
               (LEX offspring))))))
```

```
(M141!)


 CPU time : 0.00


*



;; Link proliferate to consequence for living.  Would love to have a rule do this
;; but it seems unlikely.
;; This is linked because it is a possibly correct consequence.
(describe (assert forall ($e)
&ant (build agent *e
  act (build lex "proliferate"))
cq((build agent *e
    act (build lex "perform")
    object (build lex "actions")))))
(M142!
 (FORALL V94)
 (&ANT (P175
        (ACT (M46
              (LEX proliferate)))
        (AGENT V94)))
 (CQ (P176
      (ACT (M123
            (LEX perform)))
      (AGENT V94)
      (OBJECT (M124
               (LEX actions))))))


(M142!)


 CPU time : 0.07
```

```
*

;; Link proliferate to consequence for dying.  Would love to have a rule do this
;; but it seems unlikely
;; This is linked because it is a possibly correct consequence.
(describe (assert forall ($e)
&ant (build agent *e
  act (build lex "proliferate"))
cq((build agent *e
    act (build lex "stop")
    object (build lex "living")))))
(M143!
 (FORALL V95)
 (&ANT (P177
        (ACT (M46
              (LEX proliferate)))
        (AGENT V95)))
 (CQ (P178
      (ACT (M130
            (LEX stop)))
      (AGENT V95)
      (OBJECT (M131
               (LEX living))))))

(M143!)

 CPU time : 0.00

*

;; Link proliferate to consequence for reproducing.  Would love to have a rule do this
;; but it seems unlikely
```

```
;; This is linked because it is a possibly correct consequence.

(describe (assert forall ($e)

&ant (build agent *e

  act (build lex "proliferate"))

cq((build agent *e

    act (build lex "produce")

    object (build lex "offspring")))))

(M144!

 (FORALL V96)

 (&ANT (P179

        (ACT (M46

              (LEX proliferate)))

        (AGENT V96)))

 (CQ (P180

      (ACT (M139

            (LEX produce)))

      (AGENT V96)

      (OBJECT (M140

               (LEX offspring))))))


(M144!)


 CPU time : 0.00


*


;; The moment of truth...

^(

--> defn_verb 'proliferate)

(A (stem cell

    animal

    rabbit

    mammal

    cell)
```

```
   CAN PROLIFERATE RESULT=
   (P180
    P178
    P176
    P82
    P78)
   ENABLED BY= NIL)


 CPU time : 1.92

*


End of #p/doc/graduate/663/demos/probk.demo demonstration.


 CPU time : 7.92

*
```

Appendix F

Source code for the "proliferate" background knowledge.

```
;;; Link to the representation of the passage.
(intext "proliferate.demo")

^(defn_verb 'proliferate)


;; Define a taxonomy that includes the class for rabbits, stem cells and several parent
;; classes.  The parent class common to both is "animal"

(describe (assert subclass (build lex "rabbit")
  superclass (build lex "mammal")))

(describe (assert subclass (build lex "mammal")
  superclass (build lex "animal")))

(describe (assert subclass (build lex "cell")
  superclass (build lex "animal")))

(describe (assert subclass (build lex "stem cell")
  superclass (build lex "cell")))


;; Needed for path based inference.
(define-path class
(compose class
(kstar (compose subclass- ! superclass))))


;; Based on the above superclass/subclass scheme ( a watered down taxonomy) implement
;; specific characteristics for classes.
```

```
(describe (assert object (build lex "rabbit")
  property (build lex "cute")))

(describe (assert object (build lex "rabbit")
  property (build lex "quick")))

(describe (assert object (build lex "rabbit")
  property( build lex "quick birth rate")))

(describe (assert object (build lex "rabbit")
  property (build lex "hairy")))

(describe (assert object (build lex "rabbit")
  property (build lex "warm blooded")))

;; It was difficult to model specific characteristics of cells.  At the least
;; things that stem cells possess can be modeled.

(describe (assert possessor (build lex "stem cell")
  rel (build lex "anatomical element")
  object (build lex "cell wall")))

(describe (assert possesor (build lex "stem cell")
  rel (build lex "anatomical element")
  object  (build lex "nucleus")))

(describe (assert possessor (build lex "stem cell")
  rel (build lex "anatomical element")
  object  (build lex "protoplasm")))


;; Based on the above superclass/subclass scheme ( a watered down taxonomy) implement
;; specific actions for specific classes.  An important distinction can be made here
```

```
;; and it can be shown that though sufficient for this example, mapping actions to classes
;; represented in a taxonomical hierarchy may not always be advisable.  Discussed in report.
;;
;; Consider the action walk.  Walking crosses many different classes (lots of mammals
;; walk, lots of reptiles walk, lots of birds walk, etc.) but it is not the case
;; that all mammals walk (dolphins and whales) or all reptiles walk (snakes).  A
;; hierarchy not based on taxonomy would be needed, for instance legged animals as
;; a superclass with four legged and two legged as descending subclasses.  Actions
;; would then be mappped to the calss four legged animals, of which rabbits would
;; be either a member or possibly a subclass.
;;
;; Because stem cells and rabbits are so different (in the sense of actions both can perform
;; like living, dying, reproducing, etc.) a taxonomical hierarchy is sufficient for
;; categorizing actions.
;;

;; Categorize an action for rabbits

(describe (assert forall ($r $rs)
  ant ( (build member *r
       class (build lex "rabbit"))
           (build members *rs
       class (build lex "rabbit")))
  cq ( (build agent *r
      act (build lex "hop"))
  (build agent *rs
      act (build lex "hop")))
;; could be any number of rabbit specific actions here
))

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

;; Categorize the actions for cells
```

```
(describe (assert forall ($c $cs)
  ant ( (build member *c
       class (build lex "cell"))
           (build members *cs
       class (build lex "cell")))
  cq ( (build agent *c
      act (build lex "divide"))
  (build agent *cs
      act (build lex "divide")))
;; could be any number of cell specific actions here
))


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


;; Categorize actions common to both stem cells and rabbits.  This probably
;; ought to be realized by heirarchy structure.


(describe (assert forall ($a $as $r $rs)
  ant ( (build member *a
       class (build lex "stem cell"))
           (build members *as
       class (build lex "stem cell"))
 (build member *a
       class (build lex "rabbit"))
           (build members *as
       class (build lex "rabbit")))
  cq ( (build agent *a
      act (build lex "live"))
  (build agent *as
      act (build lex "live"))
(build agent *a
      act (build lex "die"))
```

```
  (build agent *as
      act (build lex "die"))
(build agent *a
      act (build lex "reproduce"))
  (build agent *as
      act (build lex "reproduce"))
  (build agent *r
      act (build lex "live"))
  (build agent *rs
      act (build lex "live"))
(build agent *r
      act (build lex "die"))
  (build agent *rs
      act (build lex "die"))
(build agent *r
      act (build lex "reproduce"))
  (build agent *rs
      act (build lex "reproduce")))
;; could be any number of cell specific actions here
))


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


;; There now exists five actions.  For each of these actions define consequences.


;; A consequence for hopping
(describe (assert forall $e
  ant((build agent *e
     act (build lex "hop")
))
  cq((build agent *e
    act(build mod (build lex "jumping")
      head (build lex "move"))))
```

```
))

;; Proof that the rule is in working order
(describe (deduce agent $x act(build mod (build lex "jumping")
     head (build lex "move"))))

;; A consequence for dividing
(describe (assert forall $e
  ant((build agent *e
     act (build lex "divide")
))
  cq((build agent *e
    act(build mod (build lex "evenly")
      head (build lex "splits"))))
))

;; Proof that the rule is in working order
(describe (deduce agent $x act(build mod (build lex "evenly")
     head (build lex "splits"))))

;; A consequence for living
(describe (assert forall $e
  ant((build agent *e
     act (build lex "live")
))
  cq((build agent *e
    act (build lex "perform")
    object (build lex "actions")))
))

;; Proof that the rule is in working order
(describe (deduce agent $x act(build lex "perform")))
```

```
;; A consequence for dying
(describe (assert forall $e
  ant((build agent *e
     act (build lex "die")
))
  cq((build agent *e
    act (build lex "stop")
    object (build lex "living")))
))

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


;; A consequence for reproducing
(describe (assert forall $e
  ant((build agent *e
     act (build lex "reproduce")
))
  cq((build agent *e
    act (build lex "produce")
    object (build lex "offspring")))
))


;; Link proliferate to consequence for living.  Would love to have a rule do this
;; but it seems unlikely.
;; This is linked because it is a possibly correct consequence.
(describe (assert forall ($e)
&ant (build agent *e
  act (build lex "proliferate"))
cq((build agent *e
    act (build lex "perform")
    object (build lex "actions")))))
```

```
;; Link proliferate to consequence for dying.  Would love to have a rule do this
;; but it seems unlikely
;; This is linked because it is a possibly correct consequence.
(describe (assert forall ($e)
&ant (build agent *e
  act (build lex "proliferate"))
cq((build agent *e
    act (build lex "stop")
    object (build lex "living")))))



;; Link proliferate to consequence for reproducing.  Would love to have a rule do this
;; but it seems unlikely
;; This is linked because it is a possibly correct consequence.
(describe (assert forall ($e)
&ant (build agent *e
  act (build lex "proliferate"))
cq((build agent *e
    act (build lex "produce")
    object (build lex "offspring")))))


;; The moment of truth...
^(defn_verb 'proliferate)
```