Computational Contextual Vocabulary Acquisition: A Noun Algorithm

Marc K. Broklawski
State University of New York at Buffalo
Department of Computer Science and Engineering
CSE 663: Advanced Knowledge Representation
May 3, 2002

ABSTRACT

The purpose of my work throughout the current academic year related to making Karen Ehrlich's *noun* algorithm more efficient. A great amount of effort has been made to remove all the redundancy present in the original *noun* algorithm. Additionally, several miscellaneous tasks have been completed: (1) detailed instructions for running SNePS on our local CSE machines (i.e., all graduate machines), (2) a dictionary of case frames for the *noun* algorithm (those that the algorithm actually looks for), (3) detailed instructions for actually creating a demo to be run with Ehrlich's *verb* and *noun* algorithm, (4) a illustrated version of the *noun* algorithm, and (5) recreation of several of Ehrlich's original demos. Finally, I discuss future tasks related to both changes I have made and ones that need to be made in the future.

1 INTRODUCTION

Ongoing research involving the development of a CVA curricula/strategy for middle-school and high-school students is being investigated by Rapaport and Kibby (2001), which focuses on SMET texts. This research is looking to extend and enhance an algorithm that figures out the meaning of an unknown word (nouns and verbs for now)

from context, developed by Ehrlich (1995); in which her focus was on having a reader naturally acquire unknown vocabulary. That is to acquire vocabulary without resorting to asking someone or looking it up in a dictionary. This algorithm builds a dictionary-style definition consisting of specific kinds of information such as: actions, ownership, function, and structure.

I have been slowly doing tasks throughout the current academic year related to improving and enhancing Karen Ehrlich's *noun* algorithm (Ehrlich 1995). By improving, I mean more specifically making the *noun* algorithm more efficient. The code contained a great deal of redundancy that has been eliminated. In addition, I have eliminated a particular error in the algorithm that had caused it to report back incorrect information.

During the past year, I have completed many miscellaneous tasks that were meant to facilitate the coding of several passages by a few students. I provided these students with several tools to get started with: (1) detailed instructions for running SNePS on our local CSE machines (i.e., all graduate machines), (2) a dictionary of case frames for the *noun* algorithm (those that the algorithm actually looks for), (3) detailed instructions for actually creating a demo to be run with Ehrlich's *verb* and *noun* algorithm, (4) a illustrated version of the *noun* algorithm, and (5) recreation of several of Ehrlich's original demos (those in Ehrlich 1995).

In addition to the above, during the Fall '01 semester, Professor William J. Rapaport, Scott T. Napieralski, and I derived a streamlined English version of *noun* algorithm. Scott followed up this streamlined version with a more detailed English version as well.

2 VARIOUS DETAILED INSTRUCTIONS

This section includes important running instructions for the verb and noun algorithms and SNePS on a UB CSE department machine, and detailed instructions on creating .demo files for passages.

2.1 Instructions for Running CVA Algorithm and SNePS on UB CSE Machines

- Start Lisp
  - From within Emacs or XEmacs, press M-x (the M-x is generated by pressing the Esc + x keys), then type run-old-acl.
    - Note: This will **ONLY** work if you have already added (load "cselisp.el") to your .emacs file (click here for a sample .emacs file).
  - From shell prompt, type old-acl.

- To load SNePS, at the Lisp prompt type:
  - (load "/projects/snwiz/bin/sneps") or
  - :ld /projects/snwiz/bin/sneps

- At the Lisp prompt, to load the noun algorithm, type:
  - (load "/projects/rapaport/CVA/mkb3.CVA/src/fast.code") or
  - :ld /projects/rapaport/CVA/mkb3.CVA/src/fast.code

- Now start SNePS by typing (sneps) at the Lisp prompt.

- Load your demo (CVA passage) by typing (demo "<pathname>/<filename>") at the SNePS prompt.
  - <pathname> = path to directory where demo file is found
  - <filename> = demo file name
  - Example CVA demos can be found in /projects/rapaport/CVA/mkb3.CVA/demos/
    - Refer to these demos, to see what must be included in your CVA passages to run correctly.
    - Notice that the demos load the noun algorithm; thus, if we run a demo, it is not necessary to manually load the noun algorithm as above.
    - It is also important to note that prefixing anything by a ^ at the SNePS prompt or within a demo file will allow you to call something in Lisp from SNePS.

It is important to note several things regarding the above instructions. First, I have chosen to advise students to run an old version of Allegro Common Lisp (run-old-acl or old-acl). Currently, the old version of Allegro Common Lisp (ACL) on our Computer Science and Engineering local subnet is 5.0.1 (Enterprise Edition, for SPARC). The reason for running an older version of ACL at the current time is a problem with SNePS interacting with the newer version of ACL (6.1). More specifically, the problem is that the Lisp image currently shipped with ACL is no longer case-insensitive upper. A fix is in the works by Fran Johnson (current SNWIZ), which is in beta testing right now. Once this passes beta, the new version (called by either run-acl or acl) should be used, and these instructions should be updated to reflect this change.

In addition, it is important to note that in order to run both ACL and SNePS in XEmacs or Emacs, you **must** load "cselisp.el" in your .emacs file. This file can be found in your home directory by typing "ls –al" at the shell prompt. The instructions include the following sample .emacs file (showing you how to load "cselisp.el"):

```
(custom-set-variables
'(user-mail-address "abc@zooloo.cs.buf.EDU" t)
'(query-user-mail-address nil))
(custom-set-faces)
(load "cselisp.el")
```

Note that this sample .emacs file can be found at http://www.cse.buffalo.edu/~mkb3/sample.emacs.html. Also it can be found in my CVA directory at /projects/rapaport/CVA/mkb3.CVA/instructs/sample.emacs.html.

Additionally, the *Instructions for Running CVA Algorithm and SNePS on UB CSE Machines* can be found at http://www.cse.buffalo.edu/~mkb3/sneps.cva.noun.inst.html. Also it can be found in my CVA directory at

/projects/rapaport/CVA/mkb3.CVA/instructs/sneps.cva.noun.inst.html.

2.1 Instructions for Creating a Runnable Demo using The Verb or Noun Algorithm

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

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

;;; Load all valid relations
(intext "/projects/rapaport/CVA/mkb3.CVA/rels/rels")

;;; Compose necessary paths, so that the Noun/Verb Algorithm
;;; functions correctly
(intext "/projects/rapaport/CVA/mkb3.CVA/paths/paths")

;;; Load Karen Ehrlich's Noun/Verb Algorithm
^(load "/projects/rapaport/CVA/mkb3.CVA/src/fast.code")

;;; Load Knowledge Base
(intext "/projects/rapaport/CVA/mkb3.CVA/kbs/cat.base")
```

The above is a sample "header block" that must be present at the beginning of a .demo file in order to successfully run a demo using the verb or noun algorithm. Note that the three semicolons (;;;) denote a comment and not a SNePSUL or Lisp command. The header block contains six statements, where some are **mandatory** and some are **optional**.

The first thing that the user should do is reset the network, "(resetnet t)." This removes all previously built nodes and clears all previous user-entered relations (defaults back to pre-defined SNePS relations *only*). This statement is **optional**, but is **highly recommended**. This will ensure that the user is starting with a clean slate, and facilitates discovering possible errors more easily. Other nodes that have been previously built during other demos can influence the validity of the dictionary-like definition produced

by the verb and noun algorithms.

Next, the SNIP variable *infertrace* is set to nil (thus disabling infertrace). This statement is **optional**. However, I usually choose to set this variable to nil (it is set to true by default), since it makes a longer demo more easily readable. For a further explanation of this SNIP variable, refer to the SNePS 2.5 User Manual at http://www.cse.buffalo.edu/~jsantore/snepsman.

The next thing that must be done is to load all relations that are specific to the verb or noun algorithm. This is accomplished by using the "intext" command above. The "intext" command simply reads a sequence of SNePSUL commands from a specified file (in our case, the paths file) and executes each of them. As it reads each of the SNePSUL commands, it doesn't echo them. Note that your path to Ehrlich specific relations may differ from above, depending on the location of the rels file. The rels file can be found in my CVA directory at /projects/rapaport/CVA/mkb3.CVA/rels/rels. This statement is **mandatory** for the verb and noun algorithms to properly function.

In addition to loading the rels, it is also **mandatory** to load certain paths that will be used for path-based inference within the verb and noun algorithms. Similarly to loading the relations, we use the "intext" command to load the paths. These paths are important since loading them will ensure that the verb and noun algorithms are correctly functioning. Again, note that your path to Ehrlich specific paths may differ from above, depending on the location of the paths file. The paths file can be found in my CVA directory at /projects/rapaport/CVA/mkb3.CVA/paths/paths.

Next, it is absolutely **mandatory** to load the verb or noun algorithm. This is done

with the Lisp "load" function. Since the demo is running inside SNePS, we can initiate a Lisp call by preceding the "load" function with a carat (^). It is also recommended that the new fast KE verb or noun algorithm be used (fast.code), as opposed to the slower, original verb or noun algorithm (code). This new fast algorithm will be described in **Section 4**, below.

Finally, if a knowledge base is needed for the demo, you can use the "intext" command to load it. This is **optional** and depends on whether you require a knowledge base with your demo. Note that your path to the knowledge base may differ from above, depending on the location of your knowledge-base file. All knowledge-base files are appended with a .base (indicating that it is a knowledge-base file). I suggest that, if your demo is called "cat.demo" and you require a knowledge base, you call it "cat.base". The knowledge-base files for our sample demos can be found in my CVA directory at /projects/rapaport/CVA/mkb3.CVA/kbs/<filename> (where <filename> is the name of the knowledge-base file).

After the "header block," you can begin representing your passage containing some unknown word. As an example, look at the cat.demo in my CVA directory at /projects/rapaport/CVA/mkb3.CVA/demos/cat.demo. You will find many other demos in here as well.

These complete instructions can be found at http://www.cse.buffalo.edu/~mkb3/create.demos.html. In addition, they can be found in my CVA directory at /projects/rapaport/CVA/mkb3.CVA/instructs/create.demos.html.

3 A Dictionary of CVA SNePS Case Frames

I've designed a case-frame dictionary that includes all the case frames that the *noun* algorithm searches for when asked to define a noun. Since, as of now, these are the only case frames the algorithm looks for, diligence should be used when coding passages (where the noun is the unknown word) to stay within these case-frame restraints. The knowledge engineer is restricted to these thirteen case frames:

1) agent/act,
2) agent/act/object,
3) agent/act/onto,
4) antonym/antonym,
5) member/class,
6) mode/object,
7) object/proper-name,
8) object/property,
9) object/rel/possessor,
10) object1/rel/object2,
11) objects1/rel/object2,
12) subclass/superclass, and
13) synonym/synonym.

A complete listing of the syntax/semantics for these case frames can be found in the *KE Case Frame Dictionary* at http://www.cse.buffalo.edu/~mkb3/case.frame.dictionary/case.frame.index.html. In addition, it can also be found in my CVA directory at /projects/rapaport/CVA/mkb3.CVA/dictionary/case.frame.index.html.

3.1 agent/act

agent/act

Syntax:

(build agent i act j)

Semantics:

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

Sample Context:

Joe sleeps.

(describe (assert object #Joe proper-name (build lex "Joe")))

(describe (assert agent *Joe act (build lex "sleep")))

3.2 agent/act/object

agent/act/object

Syntax:

(build agent i act j object k)

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

Sample Context:

Joe hits the ball.

(describe (assert object #Joe proper-name (build lex "Joe")))

(describe (assert member #Ball class (build lex "ball")))

(describe (assert agent *Joe act (build lex "hit") object *Ball))

3.3 agent/act/onto

agent/act/onto

Syntax:

(build agent i act j onto k)

Semantics:

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

Sample Context:

King Arthur leaped on a table.

(describe (assert object #KA proper-name (build lex "King Arthur ")))

(describe (assert member #TB class (build lex "table")))

(describe (assert agent *KA act (build lex "leap") onto *TB))

3.4 antonym/antonym

antonym/antonym

Syntax:

(build antonym i antonym j)

Semantics:

[[m]] is the proposition that [[i]] and [[j]] are antonyms.

Sample Context:

"Hot" and "cold" are antonyms.

(describe (assert antonym (build lex "hot") antonym (build lex "cold")))

3.5 member/class

member/class

Syntax:

(assert member i class j)

Semantics:

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

Sample Context:

Evelyn is a person.

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

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

Required Usage:

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

3.6 mode/object

mode/object

Syntax:

(build mode i object j)

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

Sample Context:

Presumably, Joe is smart.

(describe (assert object #Joe proper-name (build lex "Joe")))

(describe (assert mode (build lex "presumably") object (build object *Joe property (build lex "smart"))))

Additional Modal Concepts:

Possibly, Joe is smart.

Sir Joe must bring the hart to the hall.

Sir Joe shall bring the hart to the hall.

All knights who wished to joust might joust.

Sir Joe can enter the lodge.

Sir Joe should not joust against Sir Marc.

Merlin said that King Arthur ought to slay his brachet.

qpossibly => sometimes automatically added when doing belief revision (e.g., smite demo)

3.7 object/proper-name

object/proper-name

Syntax:

(assert object i proper-name j)

Semantics:

[[m]] is the proposition that [[i]] has the proper name [[j]].

Sample Context:

Jack is heavy.

(describe (assert object #Jack proper-name (build lex "Jack")))

(describe (assert object *Jack property (build lex "heavy")))

3.8 object/property

object/property

Syntax:

(assert object i property j)

Semantics:

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

Sample Context:

Jack is heavy.

(describe (assert object #Jack proper-name (build lex "Jack")))

(describe (assert object *Jack property (build lex "heavy")))

3.9 object/rel/possessor

object/rel/possessor

Syntax:

(assert object i rel j possessor k)

Semantics:

[[m]] is the proposition that [[i]] is [[k]]'s [[j]].

Sample Context:

Pyewacket is Evelyn's cat

(describe (assert object #Pye proper-name (build lex "Pyewacket")))

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

(describe (assert object *Pye rel (build lex "cat") possessor *Eve))

3.10 object1/rel/object2

object1/rel/object2

Syntax:

(assert object1 i rel j object2 k)

Semantics:

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

Sample Context:

Jack is next to Fred.

(describe (assert object #Jack proper-name (build lex "Jack")))

(describe (assert object #Fred proper-name (build lex "Fred")))

(describe (assert object1 *Jack rel (build lex "next to") object2 *Fred))

Special Usage (class inclusion):

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

3.11 objects1/rel/object2

objects1/rel/object2

Syntax:

(assert objects1 i rel j object2 k)

Semantics:

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

Sample Context:

Knights varled their shields.

(describe (assert agent #KS act (build lex "varl") object #SH time #KVS))

(describe (assert objects1 *KS rel (build lex "are") object2 (build lex "knight")))

(describe (assert members *SH class (build lex "shield")))

Special Usage:

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

Reference:

Sung-Hye Cho. Representations of collections in a porpositional semantic network. In *Working Notes of the AAAI 1992 Spring Symposium on Propositional Knowledge Representation*. AAAI, March 1992.

3.12 subclass/superclass

subclass/superclass

Syntax:

(assert subclass i superclass j)

Semantics:
[[m]] is the proposition that the class of [[i]]'s is a subclass of the class of [[j]]'s.

Sample Context:

Hounds are dogs.

(describe (assert subclass (build lex "hound") superclass (build lex "dog")))

3.13 synonym/synonym

synonym/synonym

Syntax:

(build synonym i synonym j)

Semantics:
[[m]] is the proposition that [[i]] and [[j]] are synonyms.

Sample Context:

"Small" and "little" are synonyms.

(describe (assert synonym (build lex "small") synonym (build lex "little")))

4 A MORE EFFICENT NOUN ALGORITHM

I have made the original *noun* algorithm more efficient. It seems that Ehrlich, in some of her Lisp functions, had some redundancy in her code. For instance, say we have a function called "foo." In "foo," she would have a conditional that tested whether or not a path could be found. If the conditional proved to be true, she would then again search for the same path and add it to some relevant list. To alleviate the redundancy, which had an influence on slowing down the algorithm reporting back the definition given some noun as input, the choice was made to save the path to a Lisp variable while inside the conditional. If this variable turns out to contain something other then nil, we can just add it to the relevant list without searching for it again. Take a look at the following snippet of code before such changes were made and then after:

```
;definite rule, or-entail, basic-ctgy, transitive, basic
 object
    (cond ((AND #3! ((find (compose lex- act- cq- ! ant
                                    class lex) ~noun
                           (compose lex- act- agent
                                    member- class lex)
                                    ~noun))
                #3! ((find (compose lex- class- member
                                    object- cq- ! ant
                                    class lex) ~noun)))
          (list #3! ((find (compose lex- act- cq- ! ant
                                     class lex) ~noun
                           (compose lex- act- agent
                                    member- class lex)
                                    ~noun))
                #3! ((find (compose lex- class- member
                                    object- cq- ! ant
                                    class lex) ~noun))))
```

Before

```
;definite rule, or-entail, basic-ctgy, transitive, basic
 object
  (cond ((AND (setq part1 #3! ((find (compose lex- act
                                             cq- ! ant
                                             class lex)
                                               ~noun
                                    (compose lex- act-
                                             agent
                                             member-
                                             class lex)
                                               ~noun)))
              (setq part2 #3! ((find (compose lex- class
                                             member
                                             object- cq-
                                             ! ant class
                                             lex)
                                               ~noun))))
         (list part1 part2))
```

After

The new fast code can be found in my CVA directory at /projects/rapaport/CVA/mkb3.CVA/src/fast.code. I've also left the old code in the same directory at /projects/rapaport/CVA/mkb3.CVA/src/code.

In order to aid future workers who need to work on this new fast algorithm, I have created an illustrated version. I've tried to attach the actual path that each "find" looks for in the *noun* algorithm, in pictorial format, as a hyperlink. This is quite huge, so I've placed it in my CVA directory at /projects/rapaport/CVA/mkb3.CVA/docs/code.fast.pictoral.html. I should also give credit where credit is due to Scott T. Napieralski for allowing me to use his previously created pictures of these paths.

Although I have succeeded in making the *noun* algorithm more efficient, my solution has continued on some poor Lisp style by Ehrlich pointed out by Professor Shapiro. The Lisp setq's that I have used to eliminate most (if not all) of the redundancy could be a *major* source of problems in the future. This is because setq's define global variables, and, since these variables are never used anywhere else in the program, they should really be defined as local variables using the Lisp let function. If someone else in the future works on this code and is unaware of these global variables, they could wreak havoc on the algorithm. I consider this a very big problem that should be fixed as soon as possible.

5 EHRLICH'S DEMOS

Currently, I have been successful in re-creating several of Ehrlich's original dissertation demos from Ehrlich 1995. The completed demos include cat, stender, brachet, and tomato. In order to run these demos, we need some sort of background information, so I also re-created the relevant background information files for each of these demos. In order to differentiate the demo files from the background files, I label demo files as <filename>.demo and background files as <filename>.base (where <filename> equals any filename). All the demo files can be found in my CVA directory at /projects/rapaport/CVA/mkb3.CVA/demos/. In addition, the background files can be found at /projects/rapaport/CVA/mkb3.CVA/kbs/.

The completed demos were re-created by comparing the abbreviated sample runs that are present in Ehrlich 1995, and by carefully looking at the full version of her sample runs in Appendix 2 of Ehrlich 1995. This appendix can **only** be found in a computer file that she refers to in her dissertation (this appendix file is called appendix.two). The

computer file can be found in my CVA directory at /projects/rapaport/CVA/mkb3.CVA/docs/appendix.two. Please be aware that appendix.two at places seems to have been corrupted over the years. It is suggested that anyone trying to re-create additional dissertation demos be really careful when looking through this file. I would use the abbreviated sample runs that are present in Ehrlich 1995 as a guide to what the expected results of each demo are, and appendix.two only as a supplement.

Currently, I am working on re-creating the hackney demo (of Ehrlich 1995), but, due to appendix.two being corrupted, I have not been able to get the expected results that are present in the abbreviated sample runs (in Ehrlich 1995). We have thus come to the conclusion that the only way to proceed from here is to begin to recreate the hackney demo by ourselves. This should be complete in the near future.

Finally, it is also **highly** suggested that the outnet and innet functions of SNePS not be used. For example, since, at the beginning of the brachet demo run, all previous network structures built in the cat demo should be present (including background information), it should be the case that at the end of the cat demo the existing network should be outnetted to a file. So, if we wanted to run the brachet demo a few days later, then we would not have to run the entire cat demo again before running the brachet demo. We could simply innet the network that was outnetted during the completion of the cat demo run, and begin running the brachet demo. However, this seems to somehow significantly degrade the speed at which any type of deductions are done when reading through any subsequent demos (in fact, sometimes a deduction will completely hang up the system). So the moral of the story, for now, is that you will need to run all previous

demo runs that a current demo is dependent on in Ehrlich's demos. However, it has currently been discussed during research meetings with Professor Rapaport that it may not be necessary for Ehrlich's demos to be dependent on previous runs to function correctly (give the right results). Note that this is pure speculation, and a further investigation must be done to see if removing demo dependencies makes any difference.

6 STREAMLINED AND MORE DETAILED ENGLISH VERSION OF KE NOUN ALGORITHM

During the fall semester of 2001, my first task, along with Professor William J. Rapaport

```
defn_noun(N):  generate a definition for a noun, N

---------------------------------------------------------------------------------------------

Report the following information about Ns, if known:

1.  class membership, else names of individuals:

        a)  in general, report the most specific class of any class
            hierarchies

                * exception:
                        always report basic-level class membership
                * possible exception:
                        always report animal/plant/...
                        maybe report abstract/physical object

        b)  if there are no known class memberships,
                then report names of individuals who are Ns

2.  properties, else possible properties, of Ns:

        a)  if there are no known class memberships
            (i.e., if only individual Ns are known)
                then report their properties as "possible properties" of Ns

3.  structural, else possible structural, information

4.  functions, else possible functions, of Ns

5.  acts, else possible acts, that Ns perform

6.  agents that perform acts on Ns, and the acts they perform

7.  ownership information

8.  synonyms
```

and Scott T. Napieralski, was to develop a streamlined English version of the *noun*

algorithm. The hope was that, while creating such a version, we would learn how the algorithm worked and be able to supply the education members of our research group with a beginning point in developing a curriculum to eventually be taught to students.

The above is the streamlined version that was developed.

In addition, Scott T. Napieralski developed a more detailed English version of the *noun* algorithm. Since this algorithm is around 4 pages, it is included in my CVA directory at /projects/rapaport/CVA/mkb3.CVA/docs/Algorithm-v1.0.pdf. Additionally, the streamlined version above can be found in my CVA directory at /projects/rapaport/CVA/mkb3.CVA/docs/Streamlined-v1.0.pdf.

7 FUTURE WORK

One thing that probably should be done is to extend the noun algorithm to recognize the part/whole case frame. This case frame could be used to report more information in the unified dictionary frame in the slot for structure.

Also it must be decided what should appear when reporting the definition of a noun. It must be decided in what circumstances to report all slots in the definition as opposed to just specific slots. Once a particular solution is arrived at, the code must be modified to reflect the decided changes. Prof. Rapaport and I have talked about adding a flag that when set to true reports all slots, and when set to false reports only specific slots (such specific slots still need to be decided on).

Another thing that can be done to the code is to add changes proposed by students in Prof. Rapaport's Spring 2002 Advanced Knowledge Representation class, although

various research must be performed to justify these changes made to the algorithm.

Next, the Lisp setq's described above in Section 4 must be changed to Lisp let's to turn global variables to local variables. The reasons are described above, although the more I've discussed such changes with Prof. Rapaport, the more we think that it may be more practical to completely rewrite the *noun* algorithm. There seem to be a variety of errors we are experiencing and a lack of style and efficiency in the current algorithm. By continuing to work on this, with the hope of further enhancing it, we may be compounding the errors already present. So it is my opinion that the algorithm should be rewritten if so chosen by Prof. Rapaport, and the above changes in this Section could be built in from scratch.

Another important thing that must be done is resurrect SNePSwD from the dead. It appears from my conversation with Prof. Rapaport, and reading Hunt and Koplas 1998 that a particular problem with trying to get SNePSwD back and running was the new updated version of Allegro Common Lisp. That is all the information I have found to give on trying to get SNePSwD up and running again. This is important, since, in order to test some of the verb demos in Ehrlich 1995, we must get the belief revision up and running once again. Although, if we chose to rewrite the algorithm, it **would** be more practical to try to incorporate a new belief revision system currently being worked on by Fran Johnson in our department (under the guidance of Prof. Stuart Shapiro) instead of using ancient ones.

In addition, the outnet and innet problem described in Section 5 should be discussed and shown to Prof. Stuart Shapiro. Hopefully, this problem can be resolved

and make the creation of demos easier.

Also, a meeting should be made with the current SNWIZ, Fran Johnson as of now, to add the current demos in my /projects/rapaport/CVA/mkb3.CVA/demos/ folder to the default SNePS demos.

REFERENCES

[1] Ehrlich, K. (1995), "Automatic Vocabulary Expansion through Narrative Context," TR 95-09 (Buffalo: SUNY Buffalo Dept. of Computer Science).

[2] Hunt, Alan & Koplas, Geoffrey D. (1998), Definitional Vocabulary Acquisition Using Natural Language Processing and a Dynamic Lexicon, Department of Computer Science, State University of New York at Buffalo.

[3] Rapaport, William J. & Kibby, Michael W. (2001), Contextual Vocabulary Acquisition: Development of a Computational Theory and Educational Curriculum, Department of Computer Science and Engineering and Department of Learning and Instruction, State University of New York at Buffalo.