CSE 740: Verb Algorithm Revision
Final Report

Lunarso Sutanto and Chris Becker

May 4, 2004

Contents

Abstract

We started this semester with a desire to work on the verb algorithm, though at the beginning we did not know how we would work on it. As it turns out, we ended up overhauling the verb algorithm entirely, aiming to make it smaller, more logically laid out and easier to comprehend. We thought out a new design paradigm that we believe will make extending the algorithm quick and easy. And we laid out the design ideas, recommend coding styles and set an example for future designers to model after.

1 Introduction

The history of Contextual Vocabulary Acquisition in the University at Buffalo started with Karen Ehrlich's original dissertation in 1995. The goal of the project is to let a computational cognitive agent figure out the meaning of words (nouns, verbs and adjectives) from context. The knowledge representation and reasoning system used by is SNePS; Cassie is the cognitive agent that uses SNePS as the system for her "mind". The second part of the goal is to extract the algorithms used in figuring out the meaning of words and using them in the classroom to teach children to do contextual vocabulary acquisition.

Contextual Vocabulary Acquisition (henceforth labelled CVA for the rest of the paper) is performed using SNePS in 3 steps:

- Setting up background knowledge and background rules
- Representing the passage
- Processing the interaction between background rules and the passage, and extracting the results

Note that there are 3 separate algorithms used for different word classes at the third step: noun algorithm, verb algorithm and adjective algorithm.

Our work in this seminar revolves mainly in the third step of the CVA process. We redesigned the verb algorithm that extracts the pertinent information about the unknown word from the interaction of the background knowledge and the passage.

2 Motivation and Goals

Much of the previous work in this area was focused on the noun algorithm. The state of the verb algorithm at the beginning of this semester was rudimentary compared to the noun algorithm. Both the authors had done some previous research work into doing CVA for verbs and both the authors had found the verb algorithm lacking in its ability to report the meaning of the unknown words. Thus begins this semester's work in rewriting the verb algorithm.

2.1 Initial Goals

At first, we started out feeling quite unclear as to what we should try to achieve this semester. Since both of us have already done some work with

defining verbs before, this time we would like to work on the algorithm itself rather than simply working with it. We are sure that there are improvements that we could make for the verb algorithm and thus contribute in a more useful manner to the CVA project overall.

Our original ideas about how we wanted to extend the verb algorithm were:

1. Allow the algorithm to categorize and classify verbs
2. Make the algorithm report actors' and objects' features

2.2 The Verb Algorithm as of Early Spring 2004

The first step was to examine the verb algorithm as it stood. The latest version (the one that we considered most feature-rich) was the algorithm modified by Sato. The features of this algorithm include:

The main motivation behind the design of the previous verb algorithm was to have it function well with information from multiple contexts. Numerous functions were meant to determine the 'most common' of some set of features. Additionally, the data structures used were designed to separate information for each instance of the unknown verb. The main features of this version of the algorithm are listed below.

a. For all agents, objects and indirect objects do the following:

  i. Report the most common superclasses from the bottom up.

  ii. Report the most common superclasses from the top down.

b. Determine the most common type of transitivity based on the arguments found for each instance of the verb's usage in the SNePS network.

c. (Sato 2003) Identify properties of the unknown verb.

d. (Sato 2003) Identify synonyms of the unknown verb.

Overall, most of the computation in this algorithm went into determining the proper generalizations of the verb and its arguments using a variety of means. The only information it really uses in this process is class membership and superclasses of the unknown verb.

However, the algorithm had some problems. It took us slightly over a week to go through the algorithm because of its sheer size. The code is well commented and its use of white space does help in understanding

it. However, the way the code is laid out and organized was rather hard to understand. This was in a large part due to the additions that the programmers made in order to allow the program to give more information when it needs to be traced by LISP.

Furthermore, we determined that in order to add new features as we originally planned meant adding code in several places in the program as opposed to simply carving out one independent section. We thought that this would get difficult to track and debug, especially since we would like to work on implementing the changes simultaneously.

2.3 Redesigning the Verb Algorithm

We played with the idea of reworking the program from the ground up for quite a while. It seems like the amount of time required to complete such an undertaking was beyond our constraints. However, we did not relish the idea of overcoming the present challenges in the (then) current version of the program either. Finally we thought that perhaps the investment we make in redesigning the code from the ground up will be worth it when we wan to add new features in the future. Also, for all we knew, perhaps the amount of time taken in rewriting the program might be just a little bit more than the time to search the old code and understand it enough to modify it!

The goals of redesigning the algorithm are simple: we want to address the flaws present in the current version of the program and we want to improve its readability and ease of use.

In redesigning our version of the algorithm, we sought to maintain the functionality of the old version while making it more efficient. One drawback is that our version currently does not differentiate between the usage of multiple instances of the unknown verb in a representation. There is, however, the proper infrastructure in place to incorporate this if needed. However, our model seems to follow closely from van Daalen-Kapteijns and Elshout-Mohr who posit the idea that a reader employs a model of an unknown word which specifies how they incorporate new information and how they apply the information in the model to additional uses of the unknown word. As such, our algorithm may represent a similar model in that it determines what information in all the represented contexts combined should or shouldn't be used.

2.3.1 Readability

We do not wish for anyone working on the algorithm in the future to spend as much time as we did in learning the way the program works. To this end, we:

- focused on making the flow of the program be as logical as possible
- ensured that future programmers would have to read as little code as possible to get started
- added a lot of comments and explanations within the program
- cut down a lot of redundancies
- documenting our design thoroughly and pointing out clearly what parts a programmer would need to know

We hope that these features would allow others coming after us to have an easier time working with the verb algorithm.

2.3.2 Modularity

Another way to improve the readability was to make the program design modular. This meant that functions are separated logically and that adding new features does not mean having to add code in several different places at once. Instead, new additions could be placed in just one independent module. The main program that loads up the modules can run any combination of modules without them running into each other or getting hung up because they relied on each other.

This also meant that future programmers would not need to encounter modules that different authors wrote as they work on adding features to the verb algorithm. Different coding styles within a program can throw off a person trying to understand the content of the program. Instead, he can just focus on the essential parts of the program and use the modules as reference when he develops his addition.

2.3.3 Upgradeability

It should be clear by now that increased readability and a modular design are stepping stones to making the program easy to upgrade. This is quite an important design feature because it meant that future programmers can implement new features and changes as quickly as possible, thus allowing

more to be done in the scope of a semester's work. We hope that our efforts will accelerate development of UB's CVA project tremendeously.

3 Accomplishments This Semester

We are happy to report that most of our plans were carried out this semester. We rewrote the algorithm from the ground up and still had time to implement the original additions that we wanted to make. This, we believe, is a testament to how easy it is to upgrade the new version of the algorithm.

The functions implemented in the new verb algorithm and descriptions of the data structures used are in the following section. In general, the functionality of the old algorithm is completely reimplemented successfully in our algorithm. Furthermore, the way our algorithm reports transitivity is improved since the report is more complete and useful.

4 Verb Algorithm 3.0

4.1 Overall Design

The overall design of the verb algorithm is to be implemented in two parts. One part, the verb definition algorithm, written in Lisp, serves to retrieve and summarize information from the network; the second part, the inference engine, to be written in SNePSUL, will contain a generic set of rules used to make conclusions about represented actions. As of the writing of this paper, we have completed the first part of this overall design. Specifications for the second part are described in section 4.4.

The overall design of the verb definition algorithm is itself broken down into a set of modules. These modules group together into the two phases of execution of the algorithm, the first being data collection, the second being data processing and output (summarization).

Figure 4.1 illustrates the overall design of the verb definition algorithm.

4.1.1 Data Collection

Our goals for the data collection component of the verb algorithm were twofold. The first was to standardize the location of specific information in the network. For this we needed to make sure the algorithm retrieved all the relevant information present in previous verb demos. Fortunately the most recently developed verb demos followed a fairly similar design. As such, the

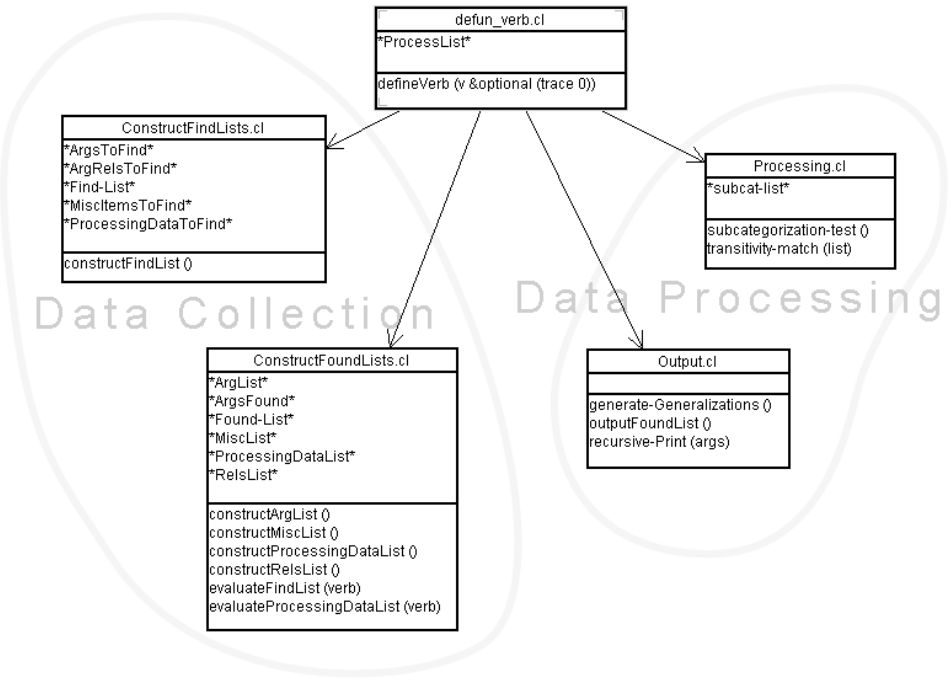

Figure 1: Overall Design of the Verb Definition Algorithm

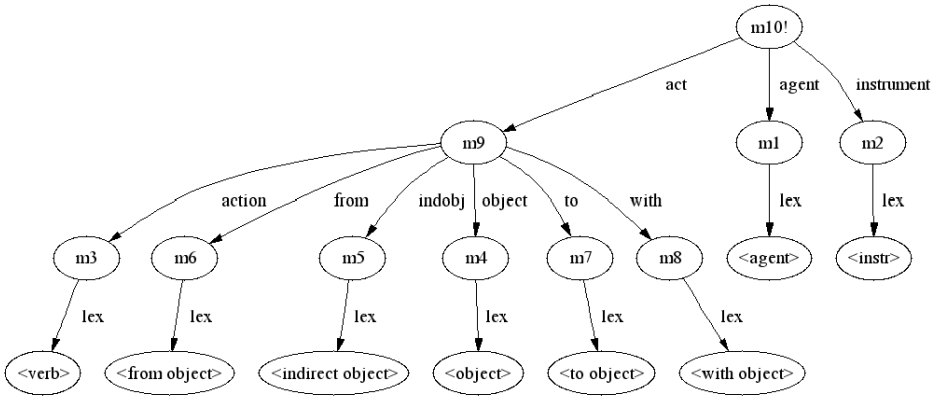

Figure 2: This illustrates the arguments of the verb that the verb algorithm currently searches for. The algorithm stores the argument nodes (e.g. m1, m2, m3, m4, m5, m6, m7, m8) in the list *ArgsFound* located in Construct-FoundLists.cl

algorithm only needs to evaluate a single path for each piece of information it needs to retrieve.

We also had to take into consideration the fact that many future verb demos might require additional case frames to best represent information in a particular context. This brings us to our second goal, which was to modularize data collection in such a way that new case frames could be added without having to modify any additional code.

As of the writing of this paper, the verb definition algorithm searches for the information illustrated in figures 4.2-4.5. This information is divided into several categories. First there are the arguments of the verb. These include object, indirect object, from-object, to-object, and with-object. It also searches for arguments of the action as a whole, agent and instrument. These are illustrated in figure 4.2. The paths for these arguments are located in the list *ArgsToFind* in ConstructFindLists.cl.

For each argument of the action and act specified above, the verb algorithm searches for the attributes property, class membership, superclass, and superordinate. The last item, superordinate, is defined as the superclass of the membership class. I included this specific path after I found it to be present in one or more verb demos. These case frames are illustrated in figure 4.3, using "agent" as the example argument. The paths for these attributes are stored in the list *ArgRelsToFind* in ConstructFindLists.cl. Upon execution, the function constructFindList, in ConstructFindLists.cl

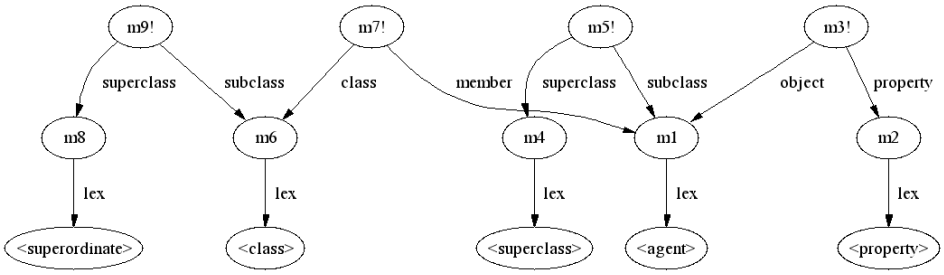

Figure 3: For each argument in figure 4.2, the algorithm retrieves the attributes illustrated here. The algorithm will store the contents of the lex nodes (e.g. superordinate,class, superclass, property) in the list *Found-List* located in ConstructFoundLists.cl

combines the elements of both *ArgsToFind* and *ArgRelsToFind* to form a single *Find-List* with the complete paths to each attribute of each argument. And example of a segment of the *Find-List* is illustrated later in this paper in figure 4.6.

In addition to the above information, the verb algorithm also searches for information independent of the specific arguments. These include case frames for cause and effect, synonymy and equivalence, and other actions performed in relation to some of the arguments of the target verb. These paths are illustrated in figures 4.3, 4.4, and 4.5.

4.1.2 Data Processing and Output

Data processing and output are two components that are closely linked since the whole idea of this part is to produce readable information. We decided it would be best to decouple these two components for the sake of flexibility and ease of modification.

Currently, the data processing component (i.e. the functions located in the file Processing.cl) consists of two functions; one to determine the argument types of the verb, and the other to determine the transitivity. We also plan on including a function to determine reflexivity of the verb, however as of now this function is not integrated into the algorithm.

The file Output.cl currently consists of two functions that return data in the list, *Found-List* in a readable format. One function simply prints out a list of all the data that was retrieved. Another function generates a set of generalized sentences using the verb in conjunction with the information retrieved for each argument of the verb.

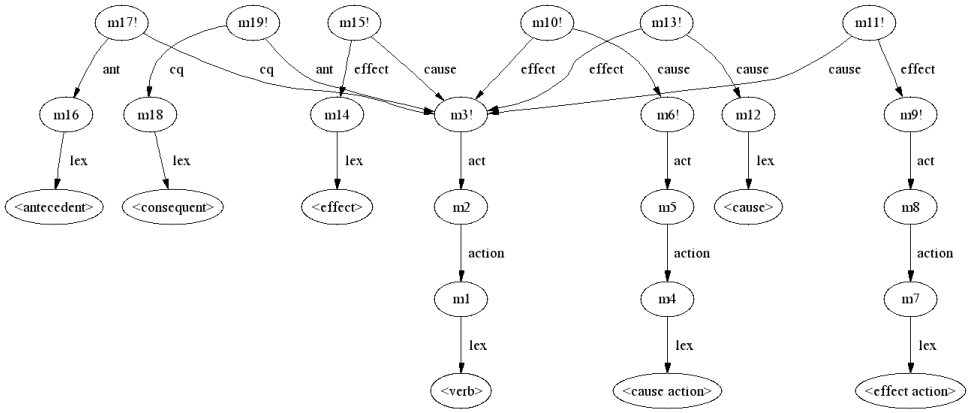

Figure 4: Illustration of the paths for cause - effect, and ant - cq that are evaluated by the verb algorithm. The algorithm will store the contents of the lex nodes (e.g. effect, cause, antecedent, consequent, cause action, effect action) in the list *Found-List* located in ConstructFoundLists.cl

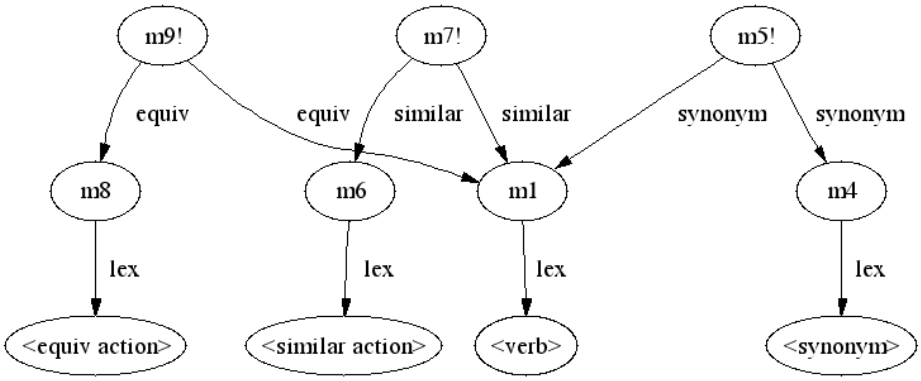

Figure 5: Paths for equiv, similar, and synonym case frames evaluated by the verb algorithm. The algorithm will store the contents of the lex nodes (e.g. equiv action, similar action, synonym) in the list *Found-List* located in ConstructFoundLists.cl

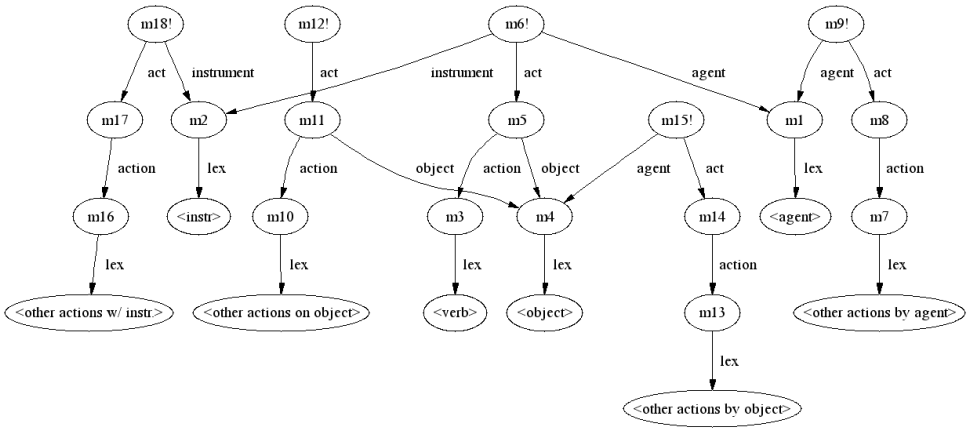

Figure 6: Actions retrieved by the verb algorithm that are related to various arguments of the verb. The algorithm will store the contents of the lex nodes (e.g. other actions w/ instr, other actions on object, other actions by object, other actions by agent) in the list *Found-List* located in ConstructFoundLists.cl

An example output of the current version of the verb algorithm appears below in figure 4.6.

```
* * * Defining the verb quail * * *

 Arguments of the verb:
   (agent)

 Transitivity:   intransitive


 * * * Basic Findings * * *

   possible synonym of quail is:
       halt,
       quail,

   possible actions performed by agent of quail is:
       quail,
       yell,
       advance,
```

```
   possible superclass of agent is:
       orcs,

 * * * generalizations from this context * * *

  Something that is a subclass of {orcs} can quail
```

4.2 Overview of the Code

The purpose of this section of the paper will be to explain the function of each component in the verb algorithm so that future students may use this as a reference when first looking at the code.

4.2.1 ConstructFindLists.cl

This file contains the initial data used by the verb algorithm to retrieve information from the SNePS network. The lists contained in this file can be added to without having to modify code in any other function. Here are the data structures:

LIST *ArgsToFind*

This list contains SNePS paths from the target verb's lex arc to the various arguments of the act and action. The following shows the content of the list.

```
(setf *ArgsToFind* (list
'(verb (lex))
'(act (action lex))
'(agent (agent- act action lex))
'(object (object- action lex))
'(indobj (indobj- action lex))
'(from (from- action lex))
'(to (to- action lex))
'(instrument (instrument- act action lex))
'(with (with- action lex))
```

As illustrated above, *ArgsToFind* is an association list with the argument name as the key and the path as the value.

LIST *ArgRelsToFind*

This list contains SNePS paths from the target verb's argument nodes to each attribute linked to it. Like the list *ArgsToFind* above, *ArgRelsToFind* is also an association list, with the attribute names as the key and the path as the value. Here is the contents of the list:

```
(setf *ArgRelsToFind* (list
'(superclass (lex- superclass- subclass))
'(membership (lex- class- member))
'(superordinate (lex- superclass- subclass class- member))
'(property (lex- property- object))
```

Upon execution, each path in *ArgRelsToFind* is appended to each path in *ArgsToFind*. This complete list is the *Find-List*, which is described in more detail later in this paper.

LIST *EvalList*

This list is not yet fully implemented. It's purpose, if needed, is to hold complex SNePSUL statements to be evaluated separately from the rest of the elements in the other lists. The following shows one possible content of the data structure:

```
(setf *EvalList* (list

'(example1 (first  (list #3! ((find (act action lex) ~verb
(act object) ~object (agent) ~agent )))))

'(example2 (first (list #3! (( find (act action lex) ~verb
(agent) ~agent))))))
```

The *EvalList* is an association list. When this is fully implemented, "example1" and "example2" would be replaced by descriptive key values that will be used to retrieve the statement to evaluate.

LIST *Find-List*

This list represents a complete list of paths which will later be iterated through and evaluated as part of a SNePS find statement. The evaluation of this is illustrated below.

```
(find (<path from *Find-List*>) verb)
```

The data returned by the find statement is then stored in the list *Found-List*, which is structured the same as the *Find-List* except that instead of paths, the cdr of each association list contains the returned data.

```
...
(agent
  ((superclass (lex- superclass- subclass agent- act action lex))
   (membership (lex- class- member agent- act action lex))
   (superordinate (lex- superclass- subclass class- member agent- act
action lex))
   (property (lex- property- object agent- act action lex))))
 (object
  ((superclass (lex- superclass- subclass object- action lex))
   (membership (lex- class- member object- action lex))
   (superordinate (lex- superclass- subclass class- member object-
action lex))
   (property (lex- property- object object- action lex))))
...
```

The structure of *Find-List* is a double association lists with the verb arguments as the key for a second association list with the attribute name as its key and full path from the verb's lex arc to the attribute's lex arc as the value.

LIST *MiscItemsToFind*

This list contains paths similar to the above lists, except the paths contained in *MiscItemsToFind* are not combined with paths from any other list. The elements of this list are merely appended to the *Find-List*. This list contains the paths for an miscellaneous items that we would need to be returned in the final output.

LIST *ProcessingDataToFind*

This list serves the same function as the *Find-List*, except that the information retrieved upon its evaluation will be stored in a separate list from the *Found-List*, which will not be returned as output, but instead just made available to the processing functions.

FUNCTION constructFindList

This function combines the contents of *ArgsToFind*, *ArgRelsToFind*, and *MiscItemsToFind* into a single list, *Find-List* which will be evaluated in the function evaluateFindList in the file ConstructFoundLists.cl. This function combines *ArgsToFind* and *ArgRelsToFind* into a double association list with the verb's argument (e.g. agent, object, etc) as the key, and another association list as the value, which contains the argument's attribute as the key and the full path from the verb's lex arc to the attribute's lex arc as the value. It also appends the list *MiscItemsToFind* to *Find-List* so that it is the central storage for all information the verb algorithm must retrieve and output.

4.2.2 ConstructFoundLists.cl

This file contains functions that use the lists from the file ConstructFindLists to actually retrieve information from the SNePS network and store it in a central data structure. This file also contains helper functions and several other global lists that assist in this process. The helper functions, which will be described in greater detail below, are used to construct lists of keys for each association list so that the values of these lists can be easily accessed. The presence of these functions allows us to modify and add to the lists in the file ConstructFindLists.cl without having to modify any additional code.

FUNCTION constructArgList
LIST *ArgList*

This function creates the list *ArgList* which contains the keys of the association list *ArgsToFind*. An example of the contents of *ArgList* is illustrated below.

```
(verb act agent object indobj from to with)
```

This list is used to iterate through in order to quickly access all the elements in the association lists *ArgsToFind*, *Find-List*, *ArgsFound*, and *Found-List*, which all use the same elements as keys.

FUNCTION constructEvalIndexList
LIST *EvalIndexList*

This function creates the list *EvalIndexList* containing the keys of the association list *EvalList*. This is used to quickly access the contents of

EvalList and *EvalData*. Note that this function is not currently in use.

FUNCTION constructMiscList
LIST *MiscList*

This function creates the list *MiscList* containing all the keys of the association list *MiscItemsToFind*. *MiscList* is used to quickly access the elements in the association lists *Find-List* and *Found-List*.

FUNCTION constructProcessingDataList
LIST *ProcessingDataList*

This function creates the list *ProcessingDataList* containing all the keys of the association list *ProcessingDataToFind*. *ProcessingDataList* is used to quickly access the elements in the association lists *ProcessingDataToFind* and *ProcessingData*. The purpose of these lists is to retrieve and store data to be used solely by any of the processing functions. These lists are currently not in use.

FUNCTION constructRelsList
LIST *RelsList*

This function creates the list *RelsList* which contains the keys of the association list *ArgRelsToFind*. The contents of this list are used to quickly access the elements of the association lists *Find-List*, *ArgRelsToFind*, and *Found-List*.

FUNCTION evaluateFindList
LIST *Found-List*

This function iterates through the *Find-List*, evaluates a find statement on the paths stored within, and stored the data returned in the association list *Found-List*. *Found-List* has the same relative structure of *Find-List* so that data can easily be retrieved.

FUNCTION evaluateProcessingDataList
LIST *ProcessingData*

This function is not fully implemented in the algorithm. It evaluates the contents of *ProcessingDataList* from the file ConstructFindLists.cl, and stores it in the association list *ProcessingData*. Unlike the contents of

Found-List, the contents of this list will not be automatically returned in the final output. Instead the data in the association list *ProcessingData* will be used by functions in the data processing component of the verb algorithm.

FUNCTION evaluateEvalList
LIST *EvalList*

This function is not complete at this time. When finished, it will evaluate the statements stored in the association list *EvalList* and store the results in the association list *EvalData*.

4.2.3 Processing.cl

This file contains functions that process the data contained in the various lists in the file ConstructFoundLists.cl. This section is currently a bit sparse, but should fill up as future students work on the verb algorithm.

FUNCTION subcategorization-test
LIST *subcat-list*

This function identifies which arguments of those stored in *ArgList* are actually arguments of the unknown verb. The types of the arguments found are stored in the list *subcat-list*.

FUNCTION transitivity-match

This function uses the data gathered in *subcat-list* to return the transitivity of the unknown verb. This is done by matching the contents of the keys in the association list match-list, depicted below.

```
(defvar match-list (list
'((agent object indobj) ditransitive)
'((agent object) transitive)
'((agent) intransitive)
))
```

FUNCTION reflexive-test

This function returns whether or not the unknown verb is used reflexively. That is, whether the value of the agent is also the value of another argument

of the verb. As of the writing of this, this function is not yet integrated into the verb algorithm code.

4.2.4 Output.cl

This file contains the output functions for the verb algorithm. Any additional processing modules that require a formatted output should have a separate output function defined in this file to do so.

FUNCTION outputFoundList

This function outputs the all the data contained in *Found-List* that was retrieved from the SNePS network. See figure 4.6 for an example output.

FUNCTION Generate-Generalizations
FUNCTION recursive-Print

These two functions work together to generate a set of generalized sentences based on the information contained in *Found-List*. These functions iterate through each type of argument in the verb's immediate context. Together they generate a sentence for each combination of attributes found for each argument. This is an expansion from the types of generalized sentences generated in the previous version of the verb algorithm. See figure 4.6 for an example output.

4.2.5 defun verb.cl

This file contains the function and list from which all other functions in the verb algorithm are called.

LIST *ProcessList*

This is perhaps the most important data structure in the verb algorithm. It contains lists of all the functions to be called, in order, for the algorithm to function. All function calls should be placed in this list.

FUNCTION defineVerb

This is the function called to define the unknown verb. It takes a single symbol as its argument and sets it to the global variable 'verb'. This function is also defined to take an optional parameter for tracing, initially set to 0. However, at this time tracing is not yet implemented. The main purpose of

this function is to loop through *ProcessList* and evaluate each element. This is what drives the entire verb algorithm.

4.3 How to add new components

4.3.1 Coding Style

The most important aspect to adding anything to the verb algorithm is to make the code readable. This means every new piece of code should be accompanied by an appropriate amount of comments explaining exactly what a particular thing does, what uses it, and what it is used for.

For commenting style, I followed the method described at the following website:

http://www.ittc.ku.edu/ hsevay/classes/eecs549/style/commenting-style-2.html

When adding a new function, be sure to provide the appropriate header comments describing the parameters the new function takes as well as its output. Also, as a courtesy to future students working on the verb algorithm, include a note in the section at the top of the file labeled "Modification History" on what was done.

4.3.2 Adding new Functions

New functions should be added in a way that best fits within the present infrastructure. That is, functions should either define a set of data to find, retrieve a set of data from a SNePS network, process previously retrieved data, or output a set of data in a particular format. Functions should be kept as decoupled as possible.

The most important aspect of the current design is that every function is called from a central location. No function should be calling other functions in the verb algorithm with the exception of that central function.

5 Sample of Results and Explanations

Complete test runs of the verb algorithm on words and contexts worked on by students in the pasts are available in the appendix. Here we present analysis on the results of those test runs.

5.1 Cripple Run

The complete results of running the new verb algorithm on the demo file for the word "cripple" by Adel Ahmed is given in the appendix. Here we would like to highlight and explain the results step by step. First off, here is the results in complete form:

```
* * * Defining the verb cripple * * *


Arguments of the verb:
  (agent object)

Transitivity:   transitive


* * * Basic Findings * * *


  possible similar action of cripple is:
      cripple,

  possible actions performed by agent of cripple is:
      cripple,
      kill,
      injure,
      kill_or_injure,

  possible actions performed on object of cripple is:
      cripple,


  possible property of verb is:
      bad,
      unknown,



  possible membership of agent is:
      windstorm,
      typhoon,
```

```
   possible property of agent is:
       violent,


   possible membership of object is:
       port,
       seaport,
       area,
       city,

   possible property of object is:
       municipal,
       large,

 * * * generalizations from this context * * *


  A {windstorm, typhoon} can cripple
 A {port, seaport, area, city}

  A {windstorm, typhoon} can cripple
 Something with the properties {municipal, large}

  Something with the properties {violent} can cripple
 A {port, seaport, area, city}

  Something with the properties {violent} can cripple
 Something with the properties {municipal, large}
nil
```

As outlined previously, the result output is formatted in a descriptive way. At the time of this writing, only the default modules are used and that is what the results above show. Basically, the algorithm will report on the transitivity of the verb. After that, it will report about case frames that it finds the verb in, the agent doing the verb in and the objects of the verb in. Finally, it will report unifications and generalizations that it can find. All other verbs will have a similar output format.

```
 * * * Defining the verb cripple * * *
```

```
Arguments of the verb:
  (agent object)

Transitivity:   transitive
```

The first section describes the transitivity of the verb. Here, it reports 'cripple' as transitive, because it finds that the verb is used within an agent-act-object case frame.

```
* * * Basic Findings * * *

  possible similar action of cripple is:
      cripple,
```

The basic findings section describes things related to the agent doing the verb, the verb itself and the object the verb is acted upon. The first thing it reports here is what other actions are similar to cripple. The algorithm will always report the verb itself, both here and in other findings later.

```
  possible actions performed by agent of cripple is:
      cripple,
      kill,
      injure,
      kill_or_injure,

  possible actions performed on object of cripple is:
      cripple,
```

Next the algorithm will report on actions perfomed on the agent and object performing 'cripple'.

```
  possible property of verb is:
      bad,
      unknown,
```

Another important part that the algorithm reports is the property of the verb. To do this, the algorithm simply looks for the object-property case frame that relates to the verb.

```
possible membership of agent is:
    windstorm,
    typhoon,

possible property of agent is:
    violent,
```

Now the algorithm reports on the agent doing 'cripple'. The pertinent information to report are the agent's class membership and property of the agent. The search for this is performed simply by looking for the subclass-superclass case frame and object-property case frame relating to the agent doing 'cripple' respectively.

```
possible membership of object is:
    port,
    seaport,
    area,
    city,

possible property of object is:
    municipal,
    large,
```

Finally, the algorithm will report about the object that 'cripple' is performed on, in a similar fashion to the report on the agent doing the verb.

```
* * * generalizations from this context * * *


 A {windstorm, typhoon} can cripple
A {port, seaport, area, city}

 A {windstorm, typhoon} can cripple
Something with the properties {municipal, large}

 Something with the properties {violent} can cripple
A {port, seaport, area, city}

 Something with the properties {violent} can cripple
Something with the properties {municipal, large}
```

Now, the algorithm will report on the unifications it can perform with the classes of things that 'cripple' can be performed on. This is probably the most interesting part for casual users of the algorithm, for it displays the result in somewhat complete sentences.

There are some things that the algorithm can report that it has not yet reported. These things come up on a need-to basis. Thus, if the algorithm cannot find the case frames that it is looking for, it will not display that incomplete result. We will see examples of these as we examine the other demos.

5.2 Fold Run

Since we have already seen an outline of the results with 'cripple', here we will only show the complete results for the verb 'fold' and point out the interesting portions of the results.

```
* * * Defining the verb fold * * *


Arguments of the verb:
  (agent)

Transitivity:   intransitive


* * * Basic Findings * * *


  possible synonym of fold is:
      go out of business,
      fold,

  possible similar action of fold is:
      fold,

  possible actions performed by agent of fold is:
      fold,
      bankrupt,
      dissolve,
      fail,
```

```
  possible property of verb is:
      unknown,
      destructive to business,



  possible superclass of agent is:
      organization,
      small business,
      business organization,

* * * generalizations from this context * * *


 Something that is a subclass of {organization, small business, business organizatio
```

'Fold' is reported here as an intransitive verb, because the algorithm can only find it used with an agent-act case frame without object. This is the only interesting difference between the run of the 'fold' verb with the 'cripple' verb.

5.3 Proliferate Run

```
* * * Defining the verb proliferate * * *


Arguments of the verb:
  (agent)

Transitivity:   intransitive


* * * Basic Findings * * *


  possible similar action of proliferate is:
      proliferate,
```

```
  possible equivalent action of proliferate is:
      proliferate,
      multiply,

  possible actions performed by agent of proliferate is:
      proliferate,
      attain,
      contaminate,

  possible actions performed with instrument of proliferate is:
      proliferate,

  possible action that is the cause of proliferate is:
      contaminate,

  possible action that is the effect of proliferate is:
      attain,

  possible membership of agent is:
      micro-organism,
      pathogen,

  possible superordinate of agent is:
      micro-organism,

* * * generalizations from this context * * *


 A {micro-organism, pathogen} can proliferate

 A {micro-organism} can proliferate
```

The new algorithm was run on the demo for the verb 'proliferate' and as can be seen, there are no substantial differences in the results for this verb as compared to the two that we have seen already.

5.4 Pry Run

```
* * * Defining the verb pry * * *
```

```
Arguments of the verb:
  (agent object indobj instrument)

Transitivity:   ditransitive


* * * Basic Findings * * *


  possible cause of pry is:
      Everything being disconnected,

  possible effect of pry is:
      Something is removed,

  possible synonym of pry is:
      remove,
      pry,

  possible similar action of pry is:
      pry,

  possible actions performed by agent of pry is:
      pry,

  possible actions performed on object of pry is:
      pry,

  possible actions performed with instrument of pry is:
      pry,




  possible membership of agent is:
      human,
```

```
  possible membership of object is:
      door panel,

  possible superordinate of object is:
      physical object,


  possible membership of indobj is:
      door,

  possible superordinate of indobj is:
      physical object,


  possible membership of instrument is:
      screwdriver,

  possible superordinate of instrument is:
      tool,

  possible property of instrument is:
      thin,
      wide,

* * * generalizations from this context * * *


 A {human} can pry
A {door panel}
A {door}
with  A {screwdriver}

 A {human} can pry
A {door panel}
A {door}
with  A {tool}

 A {human} can pry
A {door panel}
A {door}
```

```
with  Something with the properties {thin, wide}

 A {human} can pry
A {door panel}
A {physical object}
with  A {screwdriver}

 A {human} can pry
A {door panel}
A {physical object}
with  A {tool}

 A {human} can pry
A {door panel}
A {physical object}
with  Something with the properties {thin, wide}

 A {human} can pry
A {physical object}
A {door}
with  A {screwdriver}

 A {human} can pry
A {physical object}
A {door}
with  A {tool}

 A {human} can pry
A {physical object}
A {door}
with  Something with the properties {thin, wide}

 A {human} can pry
A {physical object}
A {physical object}
with  A {screwdriver}

 A {human} can pry
A {physical object}
A {physical object}
```

```
with  A {tool}

 A {human} can pry
A {physical object}
A {physical object}
with  Something with the properties {thin, wide}
```

The algorithm result on pry is more interesting than what we have seen already for the other demos. The first difference is that the verb is identified as a ditranstive verb because the algorithm sees that it was used with an agent-act-action-object-indirect object case frame. Also, here we see the use of a non-standard case-frame, which is the instrument case frame. The algorithm will treat it as something important and will report on it much like how it reports on the agents and objects of the action. Apart from this, the rest of the results are not unlike what we have seen before.

5.5 Quail Run

```
* * * Defining the verb quail * * *


Arguments of the verb:
  (agent)

Transitivity:   intransitive


* * * Basic Findings * * *


  possible synonym of quail is:
      halt,
      quail,

  possible similar action of quail is:
      quail,

  possible actions performed by agent of quail is:
      quail,
      yell,
```

```
      advance,

  possible actions performed with instrument of quail is:
      quail,




  possible superclass of agent is:
      orcs,

* * * generalizations from this context * * *


 Something that is a subclass of {orcs} can quail
```

Much of the same result output can be seen in the demo for 'quail' as the others that we have examined so far. Here the algorithm is able to pick out 'halt' as a synonym of 'quail', simply by using the synonym-synonym case frame.

6 Future Work

Since all of our goals are completed, there is not any short term goals left outstanding. Much of the future work should be focused on developing more modules and doing actual research on how to develop the algorithm further such that it can infer more useful information from the network.

6.1 Tracing

One of the reasons why the previous version of the verb algorithm was so complex was so that a trace could be done to tell exactly what processes were being called. Implementing it again on this version should not be difficult, although the value of doing so may be somewhat diminished since much of the functionality that was previously divided between several functions is now centralized into just one or two.

However, we must first look back at the reason why tracing was implemented in the first place. Basically, it was to be able to find out how the verb algorithm decided on a particular definition for a verb. For one thing, it is my opinion that this is something best answered by the inference rules

implemented in SNePS. Since the main purpose of the verb algorithm is to pull information out of the network and summarize it, there is little it can do to explain how that information came to be.

With the current state of the verb algorithm, it has some way to go before it can provide an actual definition of a verb. Right now, the information it prints out is in fact the kind of information that will tell us later on exactly how it came up with a specific definition.

6.2 Data Collection

LIST *ProcessingDataToFind*

Information not to be outputted, but rather just for processing functions to use. Currently no function uses this, and there are only two example elements in this list. However, if it is determined that a new function needs access to some additional information, then this might be of use.

Currently the infrastructure for this component is complete. The function evaluateProcessingDataList evaluates the elements of *ProcessingDataToFind* and stores them in the list *ProcessingData*.

LIST *EvalList*

This is to be used to store complex statements to evaluate, which are unable to be evaluated by the centralized evaluate functions. The *EvalList* will contain full commands that will be evaluated in the function evaluateEvalList, in ConstructFoundLists.cl. At the time of this writing, this function is not yet fully written, however the infrastructure for it is in place.

6.3 Data Processing

FUNCTION Categorize-verb

This function should categorize a verb based on associated case frames and superclass, superordinate, class membership, or other attributes of its arguments. Implementing this would involve two things. One is to set up a standard set of background knowledge containing a generic ontology for which students working on demos will link the objects of their contexts to. For a large scale standard ontology, I would suggest using WordNet:

http://www.cogsci.princeton.edu/ wn/

The second thing required to implement this function would be a data structure containing predefined verb classes and the case frames and attributes that define them. An example of such a structure is depicted below.

```
(setf *ClassList* (list
'(PTRANS1 (agent ``causal agent'') (object ``physical object'')
(to ``location''))
'(PTRANS2 (agent ``causal agent'') (from ``location'') (to ``location''))
'(PTRANS3 (agent ``causal agent'') (to ``location''))
'(MTRANS (agent ``causal agent'') (object ``abstraction'') (to ``entity''))
'(ATRANS (agent ``abstraction'') (object ``abstraction))
'(MOVE1 (agent ``causal agent'') (object ``external bodypart''))
'(MOVE2 (agent ``causal agent''))
'(MBUILD (agent ``causal agent'') (object ``abstraction''))
'(INGEST (agent ``causal agent'') (object ``substance''))
'(GRASP (agent ``causal agent'') (object ``physical object'')
(instrument ``external bodypart''))))
```

The above is an example of possible data structure for verb categorization (based on conceptual dependency)

One important thing to note regarding this possible system is that a verb may be categorized under more than one category. Additionally, any category that a verb is categorized under, based on these clues should be designated as a possible category inclusion.

6.4 Output

FUNCTION generate-moreGeneralizations

The idea behind this function would be to print more generalized sentences based on other information gathered in *Found-List*. An example of what this should output is provided below.

```
``A thing that can {<other action by agent>} can also <verb>''
``{cause/ant} can cause <agent> to <verb>''
``the result of <agent> <verb>ing <args of verb> can be {effect/cq}''
``A <agent> can <synonym of verb> <args of verb>''
``A <agent> can <equiv of verb> <args of verb>''
```

Implementing this would bring the verb algorithm another step closer to printing a good definition of an unknown verb. With more contexts and more demos, we will be able to deduce what types of information are available for creating a possible dictionary definition of a verb.

6.5 Determining Possible Verb Synonymy

One rule that may be implemented either in SNePS or in the verb algorithm should use a standard set of qualifications to compute whether one action is a possible synonym of another. One possibility is to construct it using a thresh case frame to determine whether two actions share some critical number of attributes. If that number is met then the rule will construct a synonym - synonym link between the two actions.

6.6 Determining the Relationship between Conjoined actions

Fulfilling this goal will probably require extensive research. The idea behind is to develop a rule to tell how two conjoined actions are related. Consider the following examples:

a. "It rained in Dallas and snowed in New York"

b. "I put on my shoes and went outside"

Notice that in (a) the conjoined actions are occurring simultaneously. They also share similar components (i.e. precipitation) and the arguments are of the same class (i.e. city). In a context like this we can easily determine whether two actions are similar.

Now, notice that in (b) the context represents two consecutive actions. Also, the verbs take different classes of arguments as well as different numbers of arguments. In a context such as this we can easily tell whether one action is the antecedent of another.

In both these cases the goal should be to make the resulting inference available for output by the verb algorithm.

6.7 Verb Classes

The algorithm right now recognizes the subclass-superclass caseframe for the verb itself. Thus, it will simply report whatever superclass the verb is assigned to; however, no actual inference work is actually done within the algorithm itself. What we would like to do in the future is to develop or adapt (a much likelier situation) a theory of verb classifications for use in our system. Two sources we have identified for possible adaptation are Roger Schank's Conceptual Dependency theory and Beth Levin's Verb Classes.

6.7.1 Conceptual Dependency

Roger Schank developed a theory that actions are composed of at least one primitive action. He identified a number of these primitive actions, which are listed below (taken from http://www.cse.buffalo.edu/ rapaport/676/F01/cd.html):

- PTRANS: physical transfer of location of an object
- ATRANS: abstract transfer of ownership, possession, or control of an object
- MTRANS: mental transfer of information between agents
- MBUILD: mental construction of a thought or new info between agents
- ATTEND: act of focusing attention of a sense organ toward an object
- GRASP: grasping of an object by an actor for manipulation
- PROPEL: applicaton of physical force to an object
- MOVE: movement of a body part of an agent by that agent
- INGEST: taking in of an object (food, air, water...) by an animal
- EXPEL: expulsion of an object by an animal
- SPEAK: act of producing sound, including non-communicative sounds

Classifying verbs according to these classes tell us quite a number of things about the meaning of the verb. The difficulty lies in developing a method for actually classifying the verb itself according to the information gained from the passage and background knowledge.

For example, let's examine the verb category PTRANS. The theory states that slots used for PTRANS are:

- ACTOR: a HUMAN (or animate obj) that initiates the PTRANS
- OBJECT: a PHYSICAL OBJECT that is PTRANSed
- FROM: a LOCATION at which the PTRANS begins
- TO: a LOCATION at which the PTRANS ends

The slots ACTOR and OBJECT are already defined in the standard SNePS ACTOR-ACT-ACTION-OBJECT case frame. With the new algorithm, adding the capability to recognize the FROM-TO case frame is quite trivial. In this example then, classifying a verb as a PTRANS would involve the grammar (or a human translating a passage to SNePS) using an ACTOR-ACT-ACTION-OBJECT and FROM-TO case frames where appropriate and then having a function that infers from the presence of a FROM-TO that it is actually a PTRANS. The inferring function could be implemented as a module for the algorithm or even as part of a set of standard background rules that is run against the network before the main body of the algorithm is called.

This example shows just one example of how the Conceptual Dependency theory can be developed for our project. The above example for PTRANS is rather simplistic. There are probably situations where using a FROM-TO case frame does not necessarily imply that the verb in question is a PTRANS. Similarly, there are situations where a passage using a PTRANS does not have a FROM-TO case frame. Also, we imagine that it probably would take quite a lot more work to develop similar classification methods for the other classes.

Conceptual Dependency is used widely in the field and in projects similar to ours. This gives us a wealth of knowledge in the way other researchers have circumvented problems and also suggests how we can go about developing it for our work. However, there is a problem with Conceptual Dependency literature in general: the classifications have a tendency to be revised substantially from one paper to another! Not many researchers agree on which set of rules to use. This is to a large extent due to the rather arbitrary way the primitive actions are defined. Thus, we would have to do quite a bit of research to determine what classes are going to be used and adapt the theory as necessary

subsubsectionBeth Levin's Verb Classes

Another option that we considered is based on Beth Levin's work. Her research involves studying verb usage and seeing how different verbs are used in parts of speech in similar ways. The alternations and the way verbs can be interchanged in speech can give substantial clues to their meaning.

Her work on verbs is rich. This richness and thoroughness is a two-edged sword for us: our algorithm can learn a lot more from passages; however, the amount of work required to implement the theory completely is daunting. In any case, there is a lot of information that can be used in other ways when we look at her theory.

Classifying verbs is a nice next step for our algorithm. The most important work right now is not in actually adding modules or modifying the algorithm; instead, it is more useful for someone to study and determine in what way to best develop the verb classification. With that in place, extending the algorithm would be quite an easy matter.

6.8 Other Possible Future Work

Other possibilities for the verb algorithm is difficult to define clearly at this point. Readings that we did during this semester suggests some options: knowledge of lexicon and morphology.

Zernik (CITE) created a contextual vocabulary system that he calls RINA. One important feature of this system is that it uses knowledge of how the meanings of words can change when combined with other words. He calls this knowledge of phrasal lexicon. We propose a similar method to deduce the meaning of verbs: using knowledge of the lexicon supporting the verb to figure out the meaning of the word. Right now, we do not have any further ideas about how this should be done, but this is an interesting project to do in the future nonetheless.

During our reading of Sternberg (CITE), we are reminded of a technique to figure out the meaning of words: morphology. Sternberg calls this "Decoding From Internal Context". We remember using such a technique on our GREs.

The gist of morphology is this: words can be broken down into its root, prefix and suffix. The number of roots, prefixes and suffixes is comparatively small and managable. These add features to a word. A simple example is the prefix 'un-', which for instance we use in the word 'undo' or 'uncommon'. This prefix indicates that the word means opposite of what the root means.

Such an analysis method is very powerful, since it can give us a lot of clues as to the meaning of the word. Much preliminary work though, has to be performed on the grammar translating a passage into SNePS: how do we make the grammar recognize a complex word and how to best represent it in the network. After this, then we can generate a table which the algorithm can match against to add a better understanding of the meaning of the word.

In summary, there is quite a few things that can be done to extend the verb algorithm further. The most important resource that we need right now are people willing to enlist themselves for the tasks.

Bibliography

Thorndike, Edward L. (1917), "Reading as Reasoning: A Study of Mistakes in Paragraph Reading", The Journal of Educational Psychology 8(6): 323-332

Werner, Heinz, and Kaplan, Edith (1950), "Development of Word Meaning through Verbal Context: An Experimental Study", Journal of Psychology 29: 251-257

Granger, Richard H. (1977), "Foul-Up: a Program that Figures Out Meanings of Words from Context", Proceedings of the 5th International Joint Conference on Artificial Intelligence (IJCAI-77, MIT) (Los Altos, CA: William Kaufmann): 67-68

van Daalen-Kapteijns, M.M., and Elshout-Mohr, M. (1981), "The Acquisition of Word Meanings as a Cognitive Learning Process", Journal of Verbal Learning and Verbal Behavior 20: 386-399

Sternberg, Robert J.; Powell, Janet S.; and Kaye, Daniel B. (1983), "Teaching Vocabulary-Building Skills: A Contextual Approach", in Alex Cherry Wilkinson (ed.), Classroom Computers and Cognitive Science (New York: Academic Press): 121-143

Nagy, William E., and Anderson, Richard C. (1984), "How Many Words Are There in Printed School English?", Reading Research Quarterly 19(3, Spring): 304-330

Schatz, Elinore Kress, and Baldwin, R. Scott (1986), "Context Clues Are Unreliable Predictors of Word Meanings", Reading Research Quarterly 21(4, Fall): 439-453

Johnson-Laird, Philip N. (1987), "The Mental Representation of the Meanings of Words", Cognition 25(1-2): 189-211

Zernik, Uri, and Dyer, Michael G. (1987), "The Self-Extending Phrasal Lexicon," Computational Linguistics 13: 308-327

Hastings, Peter M., and Lytinen, Steven L. (1994a), "The Ups and Downs of Lexical Acquisition", Proceedings of the 12th National Conference on Artificial Intelligence (AAAI-94, Seattle) (Menlo Park, CA: AAAI Press/MIT Press): 754-759

William J. Rapaport and Karen Ehrlich (2000). A computational theory of voculary acquisition. In L. Iwanska and Stuart C. Shapiro, editors, Natural Language Processing and Knowledge Representation, pages 347375, Menlo Park, CA. AAAI Press

William J. Rapaport and Michael Kibby (2002). Contextual vocabulary acquisition: From algorithm to curriculum.

Stuart C. Shapiro and William J. Rapaport (1987). Sneps considered as

a fully intensional propositional semantic network. The Knowledge Frontier: Essays in the Representation Knowledge.

Levin, Beth. 1993. English Verb Classes and Alternations. Chicago: Chicago University Press