Definitional Vocabulary Acquisition Using Natural Language Processing and a Dynamic Lexicon

Alan Hunt and Geoffrey D. Koplas

Department of Computer Science and SNePS Research Group
State University of New York at Buffalo, Buffalo, NY 14260-2000
`http://www.cs.buffalo.edu/sneps/`

March 31, 1998

SNeRG Technical Note 29

Definitional Vocabulary Acquisition Using Natural Language Processing and a Dynamic Lexicon

The Result of Joint Independent
Study Research with Dr. William Rapaport
SUNY Buffalo Computer Science Department
Fall, 1997

Alan Hunt and Geoffrey D. Koplas[1]

Introduction

This project is an extension of the vocabulary acquisition system developed by Karen Ehrlich as a part of her doctoral dissertation. While the system worked well, it required all of the information to be entered in SNePSUL before the system could process it. Since its stated aim was to create a system that could "read a text" and define words that it did not know from the context in which they appeared, it would be nice if the system could take natural language input rather than requiring a laborious English to SNePSUL translation. This allows a user to interact with the system without having to know anything about SNePS.

Since SNePS already has the facility to parse GATN grammars, the simplest way to create the natural language interface was to write a GATN grammar that would create the appropriate SNePS network structures for English input, and generate appropriate English responses based on the network created. This is not the first time that such a grammar was attempted - Suzanne Selib also wrote a grammar a few years ago that attempted to interface with Karen's program, but suffered several problems. These were primarily errors in the response generation section, but in our estimation, her system did not cover a general enough range of cases, and allowed little in the way of user friendly options. We decided to construct our own GATN grammar from scratch. We did, however, borrow pieces of Selib's original grammar, with credit duly given in the commented grammar code.

In the process of writing the GATN grammar, we discovered that the lexicon of words that the grammar knew how to parse was static, and using a word which was not in the lexicon would cause the parser to crash, producing no output. Since the aim of the system is to define words that are not previously known, we felt that it was important to develop a suite of functions that were capable of making the lexicon dynamic, so that new words could be added to the lexicon transparently to the user, and the parse could continue. This is not the first time that this particular project has been attempted, either, but previous versions of the code were unavailable to us, and were no longer capable of running under the current version of lisp.

The end product of our labors consists of (1) a GATN grammar, suitable to parse all sentences in Karen's original "cat" protocol, (2) a set of functions that allow the

[1] References to the authors in this and all related documentation is ordered alphabetically. Both authors contributed equally to the work presented.

lexicon to be dynamically modified, both adding words to the lexicon and changing lexical properties of existing words, and (3) the "dwarf" demo, a variant of Karen's "cat" protocol, which showcases our other two products.

File Locations and Properties

The files used and created for this project can all be found in the directory */projects/rapaport/700.hunt.koplas/* and in various sub-directories from that path. The lexicon was adapted from Suzanne Selib's lexicon, which was in turn adapted from Stuart Shapiro's *Bugbook* lexicon. Directories corresponding to both of those sources can be found at the above path. In addition, other files which were used from Suzanne's work can be found in the */Selib* sub-directory. The primary file from her work which was used in our project is the file named (rather non-descriptively) *file2*.[2] There is also a copy of this file in the sub-directory */src*. This file contains functions which convert the output of Karen Ehrlich's *defn_noun* function to natural language output. We were unable to determine any flaws in this file. Most of the difficulty Suzanne encountered in her efforts to provide NLP for Karen's work was the result of erroneous approaches in her network building and generation grammars for some of the more involved sentences of the cat demo.

Other contents of the */src* directory are the files *defn3* and *synfn*. These are files which must be loaded in order to perform any of Karen's demos, including those which have been adapted for NLP. Note that all necessary files are loaded automatically whenever any of the prepared demos are run from SNePS, as will be discussed later.

The directory */networks* contains the base networks for the demos. Both of these were created by entering the SNePSUL assertions which are the foundation for Karen's demos. The networks were then dumped to this directory using the *outnet* command. The two networks which are present are *brachet.net* and *cat.net* (the latter of which is also used for the dwarf NLP demo.) These networks are imported automatically when the demos are run.

The */lexicons* directory contains only one lexicon, which is called *lexicon2*. This file contains the lexical information necessary to run either the dwarf demo or the cat-nlu demo. The former demo will demonstrate the ability to add words to the lexicon, while the latter will simply illustrate the NLP interaction with Karen's original work. It is important to remember that, in order to repeatedly illustrate the dynamic lexicon, words added during the dwarf demo should be removed before the next running of that demo.

The */grammars* directory contains the grammar which we wrote to incorporate NLP into Karen's work, called *node_grammar*. It is commented extensively in those sections where the design decisions might seem confusing. Such confusing decisions were the result of a need to properly interface the natural language input to aspects of Karen's original work which did not foresee potential difficulties in NLP incorporation.

---

[2] In fairness, some of our files may have confusing or uninteresting names. However, the code makes reference to the names of the files in several areas, and renaming the files for cosmetic reasons did not seem to be a practical use of our time.

The grammar also contains a debug mode, which may be set by entering "Set debug to t." at the top parse level. The debug mode will ensure that a sample sentence is printed whenever any input sentence is parsed. We found this to be invaluable in our efforts to ensure that the proper networks were built.

The */docs* directory contains documentation relevant to the work. This includes a text copy of this paper, which was written as text and formatted on a home PC. A copy of the *Word* formatted version of the file is also included. Our apologies to those who prefer a *LaTeX* version. Also in the directory are a brief description of how to run the demos on which we worked (the file *running_demos*), as well as a copy of Suzanne Selib's original report documenting her work on the subject.

Finally, the */demos* directory contains all of the demos which were prepared relevant to Karen's original work. These include the *brachet.demo* and the *cat.demo* (neither of which use NLP) the *cat-nlu.demo* (which provides an NLP interface to Karen's original cat demo) and the *dwarf.demo* (which is a paraphrasing of the concepts in the cat-nlu.demo, but additionally shows the usage of the dynamic lexicon). Note that the last of these demos will modify the lexicon, but that it is the responsibility of the presenter to display the periodic changes to the *lexicon2* file by referencing it after each lexical change. All of these demos may be run directly from the SNePS *(demo)* command line by selecting *14* to denote an external file, and then entering */projects/rapaport/700.hunt.koplas/**filename***, where ***filename*** is one of the above four choices.

Also contained in the */demos* directory are the files which contain the assertions used for the base networks which correspond to the files in the */networks* directory. The file *brachet.only* contains the assertions which make up *brachet.net*, and a similar relationship holds between the file *cat.base* and *cat.net*.

It should be noted that all files mentioned here are set with full permissions for our group (*csgrads*). Thus, only those with such permissions may currently run these demos. On the other hand, such individuals currently have full permission, so they could write to the files as well as read from them. In addition, as will be noted in the future work section of this paper, a side-effect of the dynamic lexicon code is that the lexicon file becomes the exclusive property of the individual who logs in to run the dwarf demo. The need to chmod the *lexicon2* file each time after this demo is run is an annoyance which should be addressed in future work.

The Dwarf Demo

Introduction

This demo is an adaptation of Karen Ehrlich's original "cat" protocols. Originally, we switched the word "cat" to "dwarf" so that we could test the functioning of

our dynamic lexicon (dwarf was not in the lexicon, and cat was), but it was so much more amusing than cat, we decided to run with it. The dwarf demo uses the same background network that is associated with the cat demo, which may result in somewhat puzzling results, such as “Dwarves are like dogs.”

Setting up the demo

The file *dwarf.demo* located in the */demos* directory contains the NL input sentences that comprise the dwarf demo. You can run the demo as was discussed in the previous section of this paper.

The demo file will *innet* the background network *cat.net*. It will also load our grammar (*node_grammar*) and lexicon (*lexicon2*), and start the parser.

A look inside the demo

The following is a description of the sentences that are used in the demo, and why those sentences were chosen and what results they produce.

1. *Bob is a dwarf.*

At this point, the word dwarf should not be in the lexicon. The problem, therefore, is two-fold. First it needs to add dwarf to the lexicon, and then build some representation of the fact that Bob is a dwarf. It does this by building a member class caseframe, as per Karen's original translation of the sentence "Pyewacket is a cat". All sentences of the form "<propernoun> is a <noun>" will generate similar pieces of network.

2. *Define dwarf.*

At this point, the only thing that the system knows about the word "dwarf" is that Bob is one. It should report this, and does.

3. *Belgar is a king.*

This uses the same construction as (1). This will tell the system that the person named "Belgar" is a king. In this case, king of the dwarves, but the system doesn't know that. It also doesn't know that Belgar is a dwarf, because king is a different category as far as it's concerned.

4. *Bob is Belgar’s dwarf.*

Because Karen's system is designed to show ownership in the definition of a noun, it behooves us to make a statement of ownership, despite the non-PCness of suggesting that one dwarf can own another. We can, however, assume for this demo that we are working in a dwarven feudal system, and the king owns his subjects, of which Bob is one. It represents this in the network by building an *object-rel-possessor* caseframe, where the object is the node representing Bob, and the possessor is the node

representing Belgar, and the rel is the node representing the concept of a dwarf. All sentences of the form "<noun> is <proper-name>'s <noun>" will be treated similarly.

5. *Define dwarf.*

Now the system will know two things about dwarves, to wit : Bob is one, and they can be owned by kings. It will tell you this.

6. *Bob bears dwarvles.*

In order to showcase the "possible actions" slot in the define noun function, we need to have Bob (the only dwarf we know) do something. We chose to have him bear, because we are using the cat network for our background knowledge, and there is a rule that allows the system to infer that if something can bear, then it is a mammal, and an animal. Although Bob is a typically masculine name, we take refuge in the fact that since dwarves are imaginary creatures, perhaps it is the males who bear the young, in an axe-toting seahorse sort of way. The system builds a complicated rule to represent this sentence, the upshot of which is that if Bob bears a group of somethings, and if x belongs to that group, then it is a dwarvle. The reason that we chose the word "dwarvles" to represent young dwarves is that the morphological bit in the parser works overtime, and if you call them "dwarflings", it changes that to the root "dwarf", and doesn't add anything new to the lexicon.

7. *Dwarvles are young dwarves.*

Here the system will build a rule that says that if something is a dwarvle, then it is a dwarf, and has the property young.

8. *Define dwarf.*

9. *Dwalin is a dwarf.*

We introduce another dwarf to the system. The name Dwalin isn't in the lexicon at this point, so it will add it. The main point here was to show this, and also to show that stuff we say about Dwalin will affect the definition of dwarf, since he is one.

10. *Dwalin sleeps in a cave.*

We tell it, basically, that dwarves can sleep. The information that he sleeps in a cave is also stored, but it is never used in the definition function.
11. *Define dwarf.*

12. *Presumably, dwarves whistle.*

We now add some presumable actions to showcase this slot. Dwarves whistle, as anyone who has watched *Snow White* can attest to. It will build a bit of network to represent this, and tag it as "presumable" information, so that if there is a non-whistling dwarf, it will still be legal under the definition.

13. *Presumably dwarves hunt.*

This gets added so that we can see it get taken away when we tell the system that dwarves are predators. The fact that they hunt is unique information that will be reported temporarily. However, once dwarves are classified as predators the act of hunting is subsumed in the class, so it will not be explicitly reported.

14. *Define dwarf.*

15. *Dwarves are mammals.*

It already knows this, because we said dwarves could bear dwarvles.

16. *Dwarves are predators.*

Once we say this, it will take "hunting" out of the presumable actions list, since it is inherent in the definition of "predators" that they hunt, so the action becomes redundant.

17. *Define dwarf.*

18. *Dwarves are carnivores.*

19. *Dwarf is a basic-level category.*

The previous two inputs were already known by the system, but we include them in order to be true to Karen's original cat demo. The next call to *define* has no changes.

20. *Define dwarf.*

21. *Presumably, dwarves have axes.*

This is to showcase the structural properties slot. Of course, having an axe is more of a possession than a structural property. However, we needed a sentence to parallel the "Cats have whiskers" sentence in the cat demo. We tried using "Dwarves have beards", but the parser got the impression that the word *beard* was a variant on the root word *bear*. We settled for the somewhat objectionable structural property of a dwarf having an axe. All of the final information is reported in one last call to "Define".

The Dynamic Lexicon: what it does and how

The dynamic lexicon functions were created because it seemed foolish to expect a system to be learning a new word, and yet that word and all of its lexical properties had already been input *via* the lexicon. We decided to allow that, in the process of parsing a sentence, the grammar may encounter a word which is unknown. Because all of Karen's demos involve nouns and verbs, we wrote our grammar to be capable of adding such parts of speech to the lexicon. The future work section explains that this work can be expanded with a more thorough grammar. However, the convenient aspect is that the two

basic functions of our system can be used with any grammar (properly placed, of course) to achieve the desired result.

The *add-to-lex* function

This function serves the purpose of adding a new word to a lexicon. The currently loaded lexicon filename is fetched from the *current-lexicon-with-path* variable, and is opened as an output stream for appending. The function itself takes a word and a part of speech. The word is added to the lexicon, and the category (*ctgy*) lexical property of the word is set to be the part of speech for that word. Depending on the part of speech in question (recall currently only nouns, proper nouns, and verbs are supported), defaults are assumed for other lexical properties. The lexicon file is then closed, lexed in again, and parsing continues at the very same state, now with the word recognizable as the desired part of speech.

The convenient thing about the *add-to-lex* function is that it can be used in any grammar, as long as the writer of the grammar can determine what stage of the parsing will likely result in an unknown word of a certain part of speech. Examples of where such circumstances would arise can be seen in the source code of *node_grammar*.

The *modify-lexical-property* function

This function takes three parameters: a word, a lexical property to be modified, and a value to be assigned to that property. For example, in our system the input sentence "Note that mice is plural." will call *modify-lexical-property ("mice" num plur).* The keyword *note* is our way of indicating to the grammar that information about a lexical entry is about to be modified.

Our grammar is capable of composite uses of the function as well. For example, if "Note that mice is the plural of mouse." is input, then the function modify lexical property will be called four times, as follows:

1. *modify-lexical-property ("mice" num plur)*
2. *modify-lexical-property ("mice" sing "mouse")*
3. *modify-lexical-property ("mouse" num sing)*
4. *modify-lexical-property ("mouse" plur "mice")*

In addition, if either or both of the words "mouse" or "mice" were not in the lexicon, it (they) would be added and then modified. Although this is rather thorough, it is somewhat inefficient, as each call to the function results in a new *lexin*, where objectively we know that only one such *lexin* at the end should be necessary. Despite the fact that *modify-lexical-property* is functional (pardon the pun), its use is not currently illustrated in any of the demos related to Karen's work.

The portions of our grammar which deal with lexical modification could easily be imported into other grammars (and improved upon, hopefully). All that would be required is to copy the two functions discussed here, as well as the word arc on "note" at state *s*, and (in their entirety) all other states of the grammar which contain the word

"modify" as part of the state name. It should be obvious in looking at the grammar which portions this includes.

Caveats

Earlier in this section reference was made to the **current-lexicon-with-path** variable. We envisioned that this variable could be set globally when the first call to *lexin* is made by the user. This would require an update to the existing *englex.lisp* file within SNePS. Because it was not realistic for us to lobby to make permanent changes to this file, all of our demos involve the explicit setting of this variable to our own lexicon file (*lexicon2*). The actual setting of the variable takes place in Suzanne's *file2*, which is loaded for both of the NLP demos. If these functions were to be used in the future, however, the afore-mentioned (very minor) changes to the *englex.lisp* file would have to be made.

Issues with the Vocabulary Acquisition System

In the process of constructing a grammar that would parse natural language input and construct a network compatible with the SNePSUL that Karen uses as translations of the English sentences, we ran across a few areas of the work that we felt were either inconsistent or inefficient.

Class Inclusion

In her "cat" protocol, Karen translates the sentence "Pyewacket is a cat" into the following :

```
(M187! (KN_CAT life)
     (OBJECT1 B12)
     (OBJECT2 (M186 (LEX cat)))
     (REL ISA))
```

where B12 is the base node that represents Pyewacket. However, she translates the sentence "Evelyn is a person" into the following :

```
(M286! (CLASS (M22 (LEX person)))
     (KN_CAT life)
     (MEMBER B13))
```

where B13 is the base node that represents Evelyn. It is seemingly inconsistent that two such similar sentences should result in such different SNePSUL translations, and in fact, will not elicit the correct definitions if they are not translated that way. When asked for comment, Karen's only thought was that it might be based on whether or not the class for inclusion was a basic level category. However, both "cat" and "person" are considered basic level in the demo. We avoided the problem by simply building both structures for every sentence of that type.

"Misplaced" information

At several points in her translations, Karen included information that did not clearly follow (in any way we could understand) from the English sentence she was translating. For example, in the sentence "Frisky sleeps in an armchair", one of the network pieces she built was that *armchair* is a subclass of *chair*. In such cases, we added the unintuitive assertions to the background information network, where we felt they more appropriately belonged.

Lost information

Another aspect of Karen's translations that we had some trouble with was that she only builds network pieces that represent what she was interested in knowing about a particular sentence, and those characteristics were not always clear. For example, in the sentence "Frisky sleeps in an armchair", she builds an *AGENT ACT PLACE* caseframe, which keeps track of the fact that frisky is sleeping, and the place that frisky sleeps is an armchair, but it loses the preposition "in", so the original could have been "in front of" an armchair, or "on" an armchair, or even "inside of" an armchair. We changed this so that it also keeps track of the preposition, as we felt it would be more complete, and more amenable to extensions.

In several other places, she also ignores information that she doesn't need in her defining function, but gives no clear indication what she needs or doesn't need. In these cases, we build more network than she does in her translation, so that information is available in case the system may need it. In general, we tried to conserve all, or as much as possible, of the information in the input sentence.

Future work

Despite the fact that the project has been extended beyond the scope of Karen's original work, there remain a multitude of other issues that would serve to enhance both our project and Karen's original system.

The Dynamic Lexicon

Currently, every time the dynamic lexicon functions are used, the lexicon is rewritten and reloaded. This has two potential problems. The first "problem" is that it is not an exceptionally efficient procedure, since the grammar may modify the lexicon up to five times for an input sentence, and each time, the entire lexicon file is scanned and copied with appropriate changes. If the lexicon file grows large enough, this may produce a significant delay (however, at the moment, it is not significant compared to the amount of time it takes to define a noun). It would be nicer to be able to simply modify the hash table that stores the lexicon, and rewrite the lexicon file at the end of the session. We were unable to find access methods to the hash table to make this practical, however.

The second problem is that when the lexicon file is copied over, the original is deleted and the copy renamed. This works fine, but it gives the current user exclusive

ownership of the lexicon, which may keep other people from looking at it. One possible solution to this is to keep a permanent temporary file, and just rewrite the lexicon to the permanent file, and then write it back into the original lexicon file.

The Grammar

The grammar, while pretty good for the "cat" protocol, does not completely handle any of Karen's other protocols. We made some headway on sentence classes from the "brachet" protocol, but did not complete it. What needs to be done is to simply expand the number of sentence classes that the grammar can parse, and add the associated network building and response generation segments. Like any natural language interface, our grammar is necessarily restricted to a subset of English, and that subset could stand to be increased. I believe that it would be pretty straightforward to continue to add sentence classes to the grammar, but it is a time consuming task.

Secondly, we made no attempt to reconcile our grammar with SNePSwD, the version of SNePS that Karen uses to run her protocols which require belief revision. In point of fact, we were never even able to *run* SNePSwD, since the *UB UNIX* systems has upgraded its version of *ACL* since it was last used. The achievement of making SNePSwD capable of running and compatible with our grammar would be beneficial.

Finally, since the dynamic lexicon functions rely on the grammar to add or modify lexical entries, more work can be done on adding intelligent defaults and making the modification more "idiot proof". Currently, the grammar will only add words as nouns, proper nouns, and verbs, depending on if the unknown word is encountered in a noun phrase (and if so, is it capitalized), or whether it is encountered immediately after a noun phrase. The grammar could be modified to use some context (such as looking ahead to the next word in the sentence) to allow it to add adjectives, adverbs, etc. Also, modification takes place by alerting the system with the keyword "note" that you want it to modify some lexical property. As of now, it will only modify a particular subset of lexical properties (see the grammar for a more complete listing). In addition, the request has to be phrased very specifically, or it won't get parsed.

Conclusions

Working on this project has been, above all else, a learning experience. What began as an attempt to expand Suzanne Selib's grammar for the cat demo into the brachet demo became a study of why Suzanne made certain design decisions. These questions led to a further study of Karen's work and issues of what design decisions she made in her original system and why. It became clear to us that in order to understand the NL interface we were creating, we would need to do so from scratch.

In creating the grammar which would allow the NL interface with the cat demo, we became fixated with the need for a dynamic lexicon. This led to the birth of the dwarf demo. Although the dynamic lexicon we created is in its infancy, it was essential to building our understanding of how SNaLPS uses the grammar and lexicon in tandem. It

is our hope that some of what we have done this semester can be put to ongoing or future practical use. However, it is doubtless in any case that we have learned a great amount from this semester's participation on the vocabulary acquisition project.