Tatterdemalion: A Case Study in Adjectival Contextual Vocabulary Acquisition

Nicholas P. Schwartzmyer
CSE 740: Seminar on Contextual Vocabulary Acquisition
April 30, 2004

Abstract

The Contextual Vocabulary Acquisition (CVA) project is an interdisciplinary effort between members of SUNY at Buffalo's Departments of Learning and Instruction (LAI) and Computer Science and Engineering (CSE) to study how intelligent entities (both human and artificial) learn the meanings of new words from prior knowledge and contextual clues, but without the aide of external sources such as dictionaries. The goal of this project is the development of a curriculum to effectively teach CVA techniques, which will be based in large part on the insights gained by the CSE team's efforts in getting Cassie, a cognitive agent implemented in the SNePS semantic network, to define an unknown word with no more than appropriate background knowledge and the passage itself. This paper focuses on my work as part of the CSE team, in the representation of a passage containing the unknown word, "tatterdemalion", and the subsequent attempt at a definition. This work is based in large part on the CVA group protocol of the passage. I also discuss complications, both theoretical and practical, in attempting CVA on adjective (as "tatterdemalion" is used here as an adjective), particularly the difficulty in defining the word in absence of an official adjective algorithm. Suggestions are then made as to how my work may be refined, particularly in how to structure background knowledge, as well as what steps should be taken in the development of an acceptable adjective algorithm.

0 Contextual Vocabulary Acquisition: Project Overview

It has been argued for more than three quarters of a century that context plays a crucial role in one's comprehension of a text (for an early account, see Thorndike 1917). And while school-level vocabulary learning programs often stress rote learning techniques, it has been convincingly argued that the number of new words encountered in a school year far exceeds the number that could possibly be taught in the aforementioned manner (Nagy & Anderson 1984). This would seem to indicate that the majority of unknown lexical items a student (or any reader) must deal with occur in reading in texts. And as anyone who has ever encountered an unknown word while reading knows, it is not standard practice to look up its definition in the nearest dictionary. Furthermore, it has been argued that one does not actually store dictionary-style definitions for most words known in any sort of readily available way (Johnson-Laird 1987); rather, we carry some abstract notion of a word's semantics, with different features becoming salient in different contexts. Therefore, as has been argued for by various educators and psychologists (Nagy & Anderson 1984; Sternberg, Powell, & Kaye 1983), strategies for properly utilizing contextual clues should be taught, as this best fits our cognitive proclivities and would be most helpful in facilitating one's comprehension of a text, be it technical or otherwise.

The Contextual Vocabulary Acquisition (CVA) project is a joint effort between the University of Buffalo's Michael W. Kibby (Department of Learning & Instruction (LAI)) and William J. Rapaport (Department of Computer Science & Engineering (CSE)) that seeks to do just this. Its intent is to study the acquisition of an unknown word's meaning by means of contextual clues and background knowledge, but without the aid of external sources such as dictionaries or other members of the language community. Its ultimate goal is to develop a curriculum that effectively teaches CVA techniques, thus increasing reading comprehension.

The CVA project is a fully integrated interdisciplinary approach that works as follows. The LAI team collects verbal “think-aloud” protocols of unknown words and analyzes how readers come to a meaning for that word. This information is then passed along to the CSE team who attempt to represent these findings in the SNePS semantic network so that its cognitive agent, Cassie, using algorithms designed to deduce the meanings of nouns, verbs, and adjectives, may herself attempt to define these unknown words. With the insights gained from this fully analyzable, simulated reader, the LAI team will then attempt to build a curriculum to teach CVA, focusing upon group protocols and the strategies of the computational algorithms.

This paper is written as part of the CSE team’s work. Its purpose is to describe the representation and definition of a unknown word, “tatterdemalion”, by our computational cognitive agent, Cassie. Before I delve into the details of the passage and its representation, however, some background into our theoretical assumptions and the SNePS system seem worthwhile.

1 Computational CVA and SNePS: A Primer

1.1 A Theoretical Account of Computational Vocabulary Acquisition

As pointed out by Rapaport and Ehrlich (2000), there are but a few ways a computational natural language understanding system may go about acquiring new word meanings. One approach is to look it up in some sort of lexicon/dictionary. Another approach would be to guess a meaning and ask another intelligent entity (such a person) to verify or deny its hypothesis (for an example, see Zernik & Dyer 1987). A third approach would be a graph search in which a synonymous/near synonymous meaning could be located (see Hastings & Lytinen 1994). A fourth system, which is the one we use, depends upon no pre-established definition[1], whose network structure contains only knowledge that seems likely *a priori*[2], and where the only input from other intelligent systems is in the coding of the passage and plausible background knowledge—not in defining the word.

1.2 SNePS

The knowledge representation system we use as part of the CVA project is the *S*emantic *Ne*twork *P*rocessing *S*ystem (SNePS). SNePS is an intensional, propositional semantic network. In other words, the knowledge that is represented in SNePS is accessible at or above the level of truth conditions only. So for instance, if we wish to speak of Plato, we must use the Lisp-like SNePS user language (SNePSUL) to create some sort of proposition available to us. So for instance, we encode that Plato is a man, Plato is a member of the class of philosophers, Plato is an object with proper name "Plato", etc., but we cannot say anything about just Plato. This is because what is accessible in SNePS is propositional nodes--graph nodes created when some sort of relationship is established between two entities, an entity and a proposition, or two propositions.

Furthermore, SNePS is an intensional network. This means that SNePS's cognitive agent, Cassie, may believe two very different propositions about what is actually a singular observable (i.e. extensional) entity. This is a desirable feature if we wish to model a human reader (and in a larger sense, a human mind), as we think in an intensional manner. For instance, I may think that the dog down the street is a miserable creature when I am laying bed listening to

[1] This is a theoretically deliberate step on our part; we prescribe to Johnson-Laird's (1987) aforementioned notion that humans do not store definitions.
[2] Meaning that there need not be any synonymous target node *a la* Hastings (1994)

it bark at the moon, while at the same time I may think my neighbor Bill's Sheltie (who is unbeknownst to me the same howling dog!) is the cutest thing in the world.[3]

SNePS allows us to perform node-based and path-based reasoning using the SNePS inference package (SNIP), and SNePS belief revision (SNeBR) allows us to do deal with contradictions in the network. SNIP and SNeBR can be thought of as Cassie's thought processes. Using simple commands, we can ask her what she already knows or make her establish new connections within the network, in a sense generating a new belief. Such is the manner of our work in the CVA project.

2 CVA in SNePS

Allow me to lay out our procedure in brief before discussing details:

- Decide the part of speech of the unknown word
- Using verbal protocols, decide all necessary background knowledge
- Assert all background knowledge in SNePS using CVA case frames
- Add all information from the text into SNePS (i.e. have Cassie read the passage) using CVA case frames
- Run the appropriate definition algorithm.

The first step is rather trivial for the CSE team, as we aim to model a informed reader who presumably can figure out the part of speech of the unknown word with ease. Where the process really begins for us is in step two. Using both protocols administered by the LAI and those members of the CSE team run for their project word, we make note of all background information that a reader might use in his or her process of definition. At our present point in the

[3] For a more in depth discussion of SNePS and intensional networks see Shapiro & Rapaport (1987, 1995) or the SNePS Research Group homepage: http:// www.cse.buffalo.edu/sneps

project[4], this may include knowledge about the content of the passage, syntactic knowledge, and perhaps any prior experience with the word[5].

With all necessary background knowledge noted, we begin to code this knowledge in SNePS using SNePSUL. Technically speaking what we do is assert this information in the system using the SNePSUL command, *assert.* What this does is code the knowledge in the network as that Cassie already holds to be true, just as we know (or at least believe) certain things that hold relevant for the text at hand.

As we encode this information, great care is taken to adhere to a certain set of propositional relationships, or case frames. While SNePS gives us great liberty with what sorts of propositional relationships we may use, the definition algorithms used in the CVA project recognize only a small subset of these. Of course, exceptions must be made, but in order to produce the best definition, it is necessary to stay with bounds understood by the algorithms.

The next step is to simulate Cassie reading the passage. This is accomplished by representing all information in the passage with the SNEPSUL command, *add. Add* does as it implies; it adds information to the network but without necessarily asserting it, therefore creating another belief for Cassie. With the aid of SNIP, Cassie can come to believe new things, provided what is added agrees with some pre-established pattern in the network. This again mirrors a human reader; if what we read conforms to some prior belief, then we are likely to believe the newly encountered information, while if we have no background knowledge to confirm the information in the passage, then we gain knowledge of this information, but we need not *believe* it.

[4] It is a goal of the CSE team to one day utilize so-called "internal context" (Sternberg *et al.* 1983)—morphological information. At this point, however, since our NLP system is incomplete such would be premature.
[5] An example of this last type of knowledge would be in expanding the definition of an already known word, see the example of "to dress" in Rapaport and Ehrlich (2000), § 7.2.

The final step in this process is to run the appropriate definition algorithm. The CVA algorithms are essentially data collectors; they traverse the network looking for certain relationships involving the unknown word. When this graph search is complete, the algorithm reports the findings, which take the form of what Cassie believes the unknown word to mean.

3 Tatterdemalion: A Case Study of Computational CVA

With this theoretical, technical, and procedural basis in place, it is now possible to report upon the contributions to the CVA project made on my behalf. It was my goal to represent a passage containing the unknown word, "tatterdemalion", and based on the insights of "think-aloud" protocols, have Cassie define the word to be a negative quality or second-rate. I will describe the process following the procedural order laid out in § 2 as best as I can, but as will be noted, this word and passage pose certain problems that may involve some special consideration.

3.1 The Passage

Before any more is said, it is perhaps best to simply present the passage from which I worked:

> "Trains go almost everywhere, and tickets cost roughly two dollars an hour for first-class travel (first-class Romanian-style that is, with tatterdemalion but comfortably upholstered compartments and equally tatterdemalion but solicitous attendants.)"
>
> Tayler, J. (1997), "Transylvania Today", *The Atlantic Monthly* 279(6): 50-54. http://www.theatlantic.com/issues/97jun/transyl.htm

This passage was chosen amongst the five passages containing "tatterdemalion" (see http://www.cse.buffalo.edu/~rapaport/CVA/tatterdemalion.html for the complete set) as it seemed the most elucidating in both my personal protocol and that of the CVA group (see http://www.cse.buffalo.edu/~rapaport/CVA/tatterdemalion-protocol.html for a transcript).

Of course, as with most passages, there is information extraneous to, or much too complicated for the CVA project to wish to deal with. Therefore, passages are typically adapted for representation in SNePS, with hopefully little cost in semantic content. Dr. William Rapaport and I modified the above passage to read as follows (brackets mean the item was added):

> [Romanian] trains go everywhere [in Transylvania]. Tickets cost two dollars for Romanian-style first class travel, with tatterdemalion but comfortable compartments and tatterdemalion but solicitous attendants.

Since the notion that this passage was talking solely of trains of Romanian origins travelling within its borders, the addition of this bracketed information should not be troubling. All future references to my passage will consider the form immediately above.

3.2 The Problem of Adjectives

The first thing to notice is that the word is here being used as an adjective. This complicates matters in quite significant ways. From a theoretical perspective, adjectives pose the problem of being modifiers and not heads or topics. It is unlikely that much else in a text will shed light on an adjectival meaning, as text, like discourse, is structured around the heads of phrases and larger linguistic units or topics of a narrative/discourse and has little to say about referring expressions on their own. The best that can be hoped for is that a referent may have another referring expression that will act in a clearly synonymous/antonymous manner, or that the referent performs some action that will do the same (e.g., The *piscivorous* seal unsurprisingly *ate more fish*).

A second theoretical issue is that adjectives may occur in strings of length greater than one with a semantics that cannot fully be reduced to syntax.[6] Note the contrast between the following two examples (from Ferris 1993, p. 127):

(1a) The dark threatening clouds lay behind him
(1b) The dark and threatening clouds lay behind him

(2a) The leading Olympic marksmen all have phlegmatic personalities
*(2b) The leading and Olympic marksmen all have phlegmatic personalities

Note that the string in (1) may contain an intervening conjunction while that in (2) cannot. And while this distinction is, by convention, noted graphemically with a comma between the adjectives in (1), I do not feel that such makes explicit the distinction between the two constructions; namely, that both adjectives in (1) equally but separately modify the noun[7]:

```
mod'(ADJ) head(N) ∧ mod''(ADJ) head(N)
```

while in (2) the outer adjective modifies the NP such that:

```
                    mod(ADJ) head(NP)
where
                    NP → mod(ADJ) head(N)
```

Such is certainly a rather subtle complexity that is not present when dealing with nouns or verbs.

Adjectives also pose a practical problem to our CVA project. Since we do not have a standard adjective algorithm to use, deciding how to coax a definition from the system becomes

[6] If we include punctuation as a syntactic operator, at least in the matter of parsing, it will be seen this claim is not true. Since I do not know how the NLP system will work when integrated in the CVA project, this may in fact not be an issue, but I figured it is a worthwhile discussion either way.

[7] This may not be irrefutably true; it does seem that often times in constructions like (1) the ADJ closer to the N is taken as a more core argument. As I am thinking this, saying the sentence to myself, this may be an issue of prosody, the phonological equivalent to the use of a comma between the ADJs in (1) graphically. This, then, would also be unable to be reduced to the semantics of merely the two adjectives and the noun

an issue for the individual researcher. Some researchers (Lammert 2002; Garver 2002) wrote their own algorithms based upon which case frames were necessary in their passages. I opted for the less elegant approach of asking Cassie questions using the *deduce* command. More will be said about my strategy in the last part of this section.

3.3 Verbal Protocols: What we say is what Cassie needs to know

As was mentioned in § 3.1, the choice of my passage was based primarily on the insights of the CVA group's "think-aloud" protocol. Attempting to define "tatterdemalion" (actually "schmalion" a stand-in nonsense word) was a group exercise for some senior CVA researchers. This exercise, along with my reading of the passage, has given me both my target definition and what types of background information I needed to represent.

A Definition. Partly due to the aforementioned complexity of adjectives, along with the complexity of these passages, no overly definitive meaning was produced. The first passage was no doubt the most informative and produced responses such as "shabby", "run-down", "lesser", "tattered", "second class", and "a negative word".

Since the goal of this project is to attempt to define a word from context, and we in the CSE team our meant to simulate an actual reader, it was my goal to have Cassie define "tatterdemalion" to be some subset of these above definitions. While the ultimate goal is dictionary-like definitions, I am breaking with the course of some researchers who seek such a mannered definition on first encounter with the word. Such, I feel, violates the principle of our project. Therefore, if the present reader would like an printed definition for "tatterdemalion", it is suggested he or she consult his/her favorite dictionary, as none will be given here. Rather, it was my goal to have Cassie believe “tatterdemalion” means “second rate” and as a more general description, a “negative quality”.[8]

Background Knowledge. Looking over the CVA group protocol, two very particular sets of background are activated when reading this passage. One is sociocultural knowledge, the other is linguistic knowledge. The former can be seen as the focus of much of the group’s thinking, as they all seem to consider what sorts of things they know about Romania and first class travel. All members seem to hold the belief that Romania is “shabby” or “run-down”. In my opinion, this concept was applied in two ways. One, there is an implicit understanding that trains are part of a country’s infrastructure, and that the quality of a nation’s infrastructure is on par with that country’s socioeconomic status as a whole. So if the CVA group believes Romania is “shabby” or “second rate”, then it is likely that they believe a Romanian train will be the same. Secondly, and perhaps bordering on the linguistic, is the knowledge that the name of a nation in a modifying construction (e.g. *Romanian-style first class travel)* applies traits of that nation onto the modified object. Such provides the reader with the overall sense that the topic of this passage occurs, loosely speaking, within “shabby” universe of discourse, or at the very least with a negative connotation attached.

[8] Of course, I would not use these definitions if they were not at least close to the standard definition of “tatterdemalion”!

Geo-political/Infrastructure Rules

Below is an English and first order logic interpretation of these rules. For the actual SNePSUL representation, see the demo run in Appendix A.

I) Romania is a country
Country(Romania)

II) Romania is poor
Poor(Romania)

III) For all *x*, *y* if *x* is modified by 'Romanian' and Romania has the property *y*, then *x* inherits property *y*
$\forall x \forall y[(Romanian(x) \wedge Property(Romania, y)) \rightarrow Property(x, y)]$

IV) All countries have infrastructure
$\forall x\ [Country\ (x) \rightarrow \exists y\ Infrastructure\ of\ (x, y)]$

V) The traits of nations are transitive with respect to their infrastructure
$\forall x \forall y\ [(Country(x) \wedge Property\ (x,y)) \rightarrow \exists z(Infrastructure\ of\ (x, z) \wedge Property\ (z, y))]$

VI) Trains are infrastructure
$\forall x\ (Train(x) \rightarrow Infrastructure(x))$

VII) Trains inherit the properties of infrastructure
$\forall x \exists z \forall y \forall w\ [(Infrastructure\ of\ (x, z) \wedge Property(z, y) \wedge Train(w)) \rightarrow Property\ (w, y)]$

VIII) Parts of trains inherit the properties of trains
$\forall x \forall z \forall y\ [(Train\ (x) \wedge Part\ of\ (x, z) \wedge Property\ (x, y)) \rightarrow Property\ (z, y)]$

IX) Properties of parts of trains reflect the properties of trains
$\forall x \forall z \forall y\ [(Train\ (x) \wedge Part\ of\ (x, z) \wedge Property\ (z, y)) \rightarrow Property\ (x, y)]$

The second subset of sociocultural background knowledge involves facts about first class travel. Beliefs about first class travel include that it will be high quality, it will be comfortable, etc. Furthermore, tickets to travel first class will be expensive. Therefore, in this passage we get the sense that something is off when it is read that tickets for Romanian-style first class travel cost only two dollars. In my interpretation, first class travel is a category with non-graded membership—first class travel should be expensive, and two dollars is not expensive, so if something is called first class travel, but does not have expensive tickets, then it is not first class travel[9], and the niceties of such will not apply.

First class Rules

X) First class train travel is first class travel
$\forall x\ (FCTrainTravel(x) \rightarrow FCTravel(x))$

XI) First class travel is travel
$\forall x\ (FCTravel\ (x) \rightarrow Travel\ (x))$

XII) All doubly modified types of train travel (i.e. mod[mod (Romanian-style) head (first class)] head (train travel)]) are train travel
(I am not sure the best way to represent this is 1st order logic, so pardon the unconventional form)
$\forall x\ \forall(mod)\ \forall(head)\ [(mod\ head\ TrainTravel(x)) \rightarrow TrainTravel(x)]$

XIII) Tickets for first class travel are expensive
$\forall x\ [(Ticket(x) \wedge For\ (first\ class\ travel, x)) \rightarrow Property(x, expensive)]$

XIV) First class travel is comfortable and of high quality
$\forall x\ [FCTravel(x) \rightarrow (Property(x, comfortable) \wedge Property(x, of\ high\ quality)]$

XV) If tickets for travel cost two dollars, then they are not expensive, which means that this is not first class travel
$\forall x\ [(Ticket(x) \wedge Cost(x, two\ dollars) \rightarrow \neg\ (Property(x, expensive) \wedge For\ (first\ class\ travel, x))]$

XVI) If something is not first class travel, then it will not be high quality
(I here am using the contrapositive, as this was what I used in SNePS, as to not upset SNIP)
$\forall x\ [(Travel\ (x) \wedge Property(x, high\ quality)) \rightarrow \neg\ (FCTravel(x))]$

Linguistic knowledge is the second category of background knowledge crucial to understanding this passage. We may call it a category, but what it really boils down to is a proper interpretation of the semantics of "but". The constructions "tatterdemalion but comfortable" and "tatterdemalion but solicitous" proved to be quite informative in the CVA

[9] My proposal then, is that the constituent, Romanian-style first class travel, should follow the pattern:
mod[mod(Romanian-style) head (first class)] head (travel)
Such is, in fact, the representation used in my SNePS representations. If any future researcher wishes to claim that FC travel is graded, and that RSFC travel is still first class travel, just somehow mitigated, then he or she should *not* use the above representational schema. Use instead: mod (Romanian-style) head [mod (first class) head (travel)]

protocols, as they put the unknown word in what is instinctively an opposition with positive qualities. With the previous background knowledge giving the text a negative force, and this unknown word standing in opposition with one of the few positive traits present in the passage, these “but” constructions definitely factored in stating that “tatterdemalion” must be a negative.

The question I faced, however, is if this is actually in the semantics of “but”, and furthermore what else is it that “but” says—must it be that “tatterdemalion” is the negation of solicitous and comfortable? When one thinks about this for a moment, this is clearly not the case, so the semantics of “but” must be a more subtle thing.

My solution to the semantics of “but” follow the work of Segal and Duchan (1997). They analyzed the usage of “but’ in both juvenile and adult texts and developed a schema for how “but” is used. They call this schema the Domain-Expectation-Violation (DEV) schema. Quite simply, the preceding argument of “but” sets the domain, for which there should be certain expectations that are then violated by the argument that follows “but”. In this passage, however, we have an added complication in that we do not know what are the expectations of the domain (being that the domain is “tatterdemalion”, the unknown word). I would argue it to be a minor complication, though—it seems to me that the reader already has a negative sense of the topic, and the presence of “but comfortable” and “but solicitous” are already taken as a positive violation to the negative tone of the passage. Hence, “tatterdemalion” is merely added to the negative domain. (This is arguably a good deal of sophisticated talk, but with the same simplistic interpretation of the semantics of “but”.)

With this in hand, my treatment of “but” is as follows:

But rule

XVII) If object 1 is in a "but" relationship with object 2, and object 2 is a positive attribute, then object 1 is a negative attribute, equivalent to a negative quality[10]

$\forall x \forall y[(But(x, y) \wedge Positive\ Attributes\ (y)) \rightarrow$
$(\ Negative\ Attributes(x) \wedge Equivalent\ (x, negative\ quality) \wedge \forall z(Property(z, x) \rightarrow$
$Property(z, negative\ quality)))]$

Of course, this should appear immediately problematic, as the DEV schema is intuitively commutative, for the following is just as felicitous:

$$\forall x \forall y\ [(But\ (y, x) \wedge Positive\ Attributes\ (y)) \rightarrow Negative\ Attributes\ (x)]$$

Hence, my rule is too specific, missing a linguistic generalization. A more general "but" is one of my suggestions in §4.

Tied into the "but" rule, of course, is our need to know that "comfortable" and "solicitous" are positive attributes:

Property knowledge

XVIII) Comfortable is a positive attribute
Positive Attributes (comfortable)

XIX) Solicitous is a positive attribute
Positive Attributes (solicitous)

As for my last piece of background knowledge, I attempted to write a rule that simulates how I felt these background knowledge sets piece together. While there is no good way to represent a 'negative context', so to speak, I picked up upon a certain few crucial bits of this background knowledge that I felt were key to defining "tatterdemalion" as "second rate". First, there is what we believe non-first class travel is like. Secondly, in order to pick up on the fact

[10] The rule is busy, I know. If one looks at the SNEPUL representation, we see that I use this rule to make Cassie understand an adjective in terms of *object-property*, *equiv-equiv*, and *member-class*. Such is why this is busy; I was

that parts (i.e. attendants, compartments) of the train in the passage are what will be described as "tatterdemalion", I made reference to the "but" rule in terms of train parts.

Linguistic/Pragmatic Context Unification Rule

XX) If something is not first class travel, and trains (used for this travel) have parts whose properties are described in terms of the above "but" schema, then one of these properties will be equivalent to being second rate.

$$\forall x\, \forall y\, \forall t\, \forall v\, \forall z\, \forall w[(\neg FCTravel(x) \wedge Train(t) \wedge Part\ of(t, v) \wedge Property(v, z) \wedge Property(v, w) \wedge But\ (z, w)) \rightarrow Equivalent\ (w, second\ rate)]$$

And while, as was said before, I feel this rule simulates the thought processes used by readers in deducing the meaning of "tatterdemalion", it is not without its problems. A major concern is whether this rule has any cognitive validity in its form. Again, I feel it mirrors *what* we do, I am not so certain it best describes *how* we do it.

Smaller issues occur as well. For instance, the aforementioned issue with the "but" rule applies. Another issue is that, if one looks at the SNEPSUL code, the trains referenced are not logically linked with the idea that they are used for the non-first class travel. As I ran out of time, I left it as such, but changing this is one of the first things that need be done if one wishes to tackle this passage.

3.4 Cassie Reads the Passage

With this background knowledge in place, we now are able to add the information from the passage itself in SNePS, in essence allowing Cassie to read the passage. I will here show the actual SNePSUL representations, making comments where appropriate.

merely testing out in what manners it makes sense to talk of adjectives. As to whether this is the most logical rule is a different matter.

Passage Representation

(PR 1) Romanian trains go everywhere in Romania:

```
(describe (add agent
      (build  mod (build  lex "Romanian")
            head (build lex "train"))
       act
      (build action (build lex "go\ everywhere\ in")
             object (build lex "Transylvania"))))
```

As with most of the passage, the mod-head case frame is essential; for the syntax and semantics of this and all “non-standard” case frames, see Appendix B. The proposition itself seemed very easily fit into the agent-act-action-object frame. It would also be a possibility to use a SNePS location case frame to represent “go everywhere in Transylvania”, but as this was rather unimportant for defining “tatterdemalion”, I took this slightly less elegant approach. Any future researcher may want to consider a different representational pattern here.

(PR 2) Romanian trains are trains

```
(describe (add member
      (build mod (build lex "Romanian")
             head (build lex "train"))
         class (build lex "train")))
```

While not explicit in the passage, such a proposition does not seem to be a readily available piece of background knowledge. Rather, it seems more like something we quickly realize while reading (this will ‘prime’ background rules III-VII and XII).

(PR 3) Romanian-style first class (RSFC) trains are a subclass

```
(describe (add member
      (build mod (build mod
                  (build lex "Romanian-style")
                        head
                  (build lex "first\ class"))
            head (build lex "train") = rsfctrain)
      class
        (build mod (build lex "Romanian")
             head (build lex "train"))))
```

This was the best manner in which I felt I could introduce the concept of RSFC trains, necessary in understanding the passage, but certainly something one would not have any prior knowledge about.

(PR 4) RSFC train travel is a member of the class of first class train travel

```
(describe (add member
      (build mod
            (build mod
            (build mod (build lex "Romanian-style")
                  head (build lex "first\ class"))
                  head (build lex "train"))
             head (build lex "travel"))
       class
      *fctravel))
```
[11]

What is tricky about this passage is that the reader needs an active thought process about both RSFC trains and RSFC train-travel. I quite frankly kept confusing myself as to which I needed to talk about in terms of defining “tatterdemalion”, for it seemed both play some role. Regardless, like the concept of RSFC trains, RSFC train-travel is very unlikely something one has in their previous knowledge, so it must be added when read in the text, even this is not an explicit proposition in it.

(PR 5) Tickets for RSFC travel exist.

```
(describe (add
      object1 #ticket
      rel (build lex "for")
      object2 (build mod
                  (build mod
                        (build mod (build lex "Romanian-style")
                              head (build lex "first\ class"))
                        head (build lex "train"))
                   head (build lex "travel"))))
```

Again, tickets are not introduced in the passage in such a manner, but really little is, most

[11] The *fctravel shortcut was introduce in the background knowledge, see Appendix A.

propositions of this nature are taken for granted. I could not think of an argument as to why any other case frame would work better than *object1-rel* (“for”)-*object2*, but a future researcher may wish a different approach.

(PR 6) An object called #ticket is a member of the class of tickets

```
      (describe (add member *ticket class (build lex "tickets")))
```

I do not recall why I used a Skolem constant for tickets, but (PR 6) is need to ensure that the base node does in fact reference a ticket.

(PR 7) Tickets for RSFC travel cost two dollars

```
      (describe (add object
            (build lex "two\ dollars")
            rel
            (build lex "cost")
            possessor
            *ticket))
```

The first issue I will tackle here is the case frame I used, *object-rel-possessor*. The SNePS “rule-of-thumb” for verbs (i.e. “cost”) is the *agent-act-action-object* frame. However, as I argue in my demo annotation, since “cost” is a stative verb, I do not feel such is the best approach. Rather, thinking about how such states are treated cross-linguistically, I settled for the notion that while “cost” is here a verb, it is in fact referencing something *having* a cost, much like how age is represented in Spanish, *tengo vienticinco anos* (literally, ‘I have 25 years’).

The second, and much more interesting issue, is that this piece of added information derives a contradiction with background rule XV. This was intentional on my part. Studying the CVA protocol and thinking of my own reading it does seem that, by somewhere around this line in the passage, we must deal with the conflicting notions of what we know of first class travel, and the sense we are getting of RSFC travel. And as I argued for in § 3.3, I feel that we should not treat first class travel to be a graded category. Hence, Cassie, in adding this information is alerted by SNIP that there is a rule saying that something in the proposition of (PR 6) cannot be first class travel, but we are saying it is. This invokes SNeBR, the SNePS belief revision system, through which, following the procedure given in the demo notes,[12] we discard the hypothesis that RSFC travel is in fact first class travel, allowing a slew of our background knowledge rules that will lead to a definition of "tatterdemalion" to fire.

(PR 7) RSFC trains have compartments

```
            (describe (add    object1 *rsfctrain
                        rel (build lex "have")
                        object2 #compartments))


      (describe (add member *compartments class (build lex
                  "compartments")))
```

(PR 8) RSFC trains have attendants.

```
            (describe (add
            object1 *rsfctrain
                  rel (build lex "have")
            object2 #attendants))


            (describe (add member *attendants class (build lex
                  "attendants")))
```

The above two representations are fairly standard and would seem to need little more in the way of comments.

[12] It is worth restating in this report, in case the reader misses it in the beginning of my demo, that one must run SNePS 2.61 for a demo to work with contradiction. SNePS 2.61 is found at /projects/shapiro/Sneps/sneps261.cl

(PR 9)   These compartments are tatterdemalion

```
(describe (add object *compartments
      property (build lex "tatterdemalion")))
```

(PR 10) These compartments are comfortable

```
(describe (add object *compartments
      property (build lex "comfortable")))
```

(PR 11) Comfortable and tatterdemalion are in a “but” relationship

```
(describe (add object1 (build lex "tatterdemalion")
      rel (build lex "but")
      object2 (build lex "comfortable")))
```

This (and PR 14) will fire our “but” rule.

(PR 12) These attendants are tatterdemalion

```
(describe (add object *attendants
      property (build lex "tatterdemalion")))
```

(PR 13) These attendants are solicitous

```
(describe (add object *attendants
      property (build lex "solicitous")))
```

(PR 14) Solicitous and tatterdemalion are in a “but” relationship

```
(describe (add object1 (build lex "tatterdemalion")
      rel (build lex "but")
      object2 (build lex "solicitous")))
```

3.5 Asking Cassie What “Tatterdemalion” Means

As was stated in § 3.2, there is no official adjective algorithm in our CVA project, and I chose to get a definition by using *deduce* command. This gave me some freedom in pondering what types of information are useful in trying to define adjective via context. Such is why some of my rules seem busy, as I was experimenting with useful frames. As it turned out all manners in which I wished to a define “tatterdemalion”—equivalence, class membership, and object modified—succeeded:

```
          ; Ask Cassie what "tatterdemalion " means:
          ;==========================================


;                       Equivalences
; =====================================================


; Following our "but" rule, tatterdemalion should be equivalent
; to a negative quality:

(describe (deduce
                equiv (build lex "tatterdemalion")
                equiv (build lex "negative quality")))

(m150! (equiv (m118 (lex tatterdemalion)) (m47 (lex negative quality))))

(m150!)


; Cassie agrees!


; Following our unified linguistic/pragmatic rule,
; the meaning of "tatterdemalion" should be something like
; 'second rate':

(describe (deduce
                equiv (build lex "tatterdemalion")
                equiv (build lex "second rate")))

(m134! (equiv (m118 (lex tatterdemalion)) (m50 (lex second rate))))

(m134!)
; Cassie agrees!
;                        Class Membership
; ===============================================================

; Properties can be members of larger class of properties, e.g.
; the property 'green' is a member of the class of properties
;  we refer to as 'colors'

;  Following the protocols, the most we can say about "tatterdemalion"
;  is that it is (probably) a negative attribute, let us see
;  if Cassie agrees, following the rules given to her:

(deduce member (build lex "tatterdemalion") class (build lex "negative
attributes"))

(m151!)

;  She does.



;                        Modified Entities
;  ===============================================================

; Lets find the nodes with object- property arc with "tatterdemalion":

(dump (find (object- property) (build lex "tatterdemalion")))

(b4
 (member- (m147 m146 m144 m143 m141 p124 m117!) object-
  (m148! p122 m140 m139 m138 m137 p121 p120 m136!) object1-
  (p126 m142 p125) object2- (p127 p123 m115!) possessor- (m145)))
(b3
 (member- (m131 m130 m128 m127 m125 p116 m114!) object-
  (p112 m124 m123 m122! m121 p111 p110 m119!) object1- (p118 m126 p117)
  object2- (p119 p115 m112!) possessor- (m129)))

(b4 b3)

; And what are these entities?

(describe m114 m117)
(m117! (class (m116 (lex attendants))) (member b4))
(m114! (class (m113 (lex compartments))) (member b3))
```

Now, this method of defining “tatterdemalion” would not be our favored approach, as by asking Cassie questions outright, we our making this system less autonomous and more akin the work of Zernik and Dyer (1987). However, I would feel my part in this project somewhat of a loss if I did not get some sort of definition for the word. For, while representing textual and background knowledge is an invaluable task in terms of the CVA project's greater good, the glory is really in a good definition.

4 Future Work

4.1 Immediate Steps

The work available immediately on this passage mostly involves the fixing of present rules. As was mentioned prior, my realization of a "but" rule is too specific, and the commutative generalization of the "but" schema need be accounted for. Likewise, the Linguistic-Pragmatic Unification Rule needs the antecedent involving trains to be linked logically with the notion that these trains are used in first class travel. This would likely involve some sort of additional antecedent with a *member-class* frame, linking the variable 'train' to first class trains. There still exists a commented out antecedent in which I attempted this with the rule.

Another immediate step would be as follows. I realized when I was looking over my demo that, as where I represented the concept that Romania is poor and that anything modified by "Romanian" should be the same, I never actually applied this in defining "tatterdemalion" within SNePS. Since this figured into process for the CVA group and myself, so should it for Cassie. Therefore, a further rules(s) needs to be written to enrich the network involving "tatterdemalion"—even if it cannot enrich the definition.

4.2 Long-term Goals

On the shorter side of long-term goals, if one wishes to use some of my representations as part of his or her work, then they should consider the logical structure of some of them, notably the Linguistic-Pragmatic Context Unification Rule. As was mentioned before, the type of reasoning I am attempting to get at seems valid, but the form, even to me, seems suspect. One needs to think of a more elegant solution as to how linguistic and pragmatic contexts should be pieced together.

Another goal, of course, is to get “but” out of rule-based knowledge representation and into the lexicon. This is obviously a more intuitive place for linguistic knowledge to begin with, and it also saves one the headache of trying to accurately translate the semantics of “but” into logical rules.

Lastly, there is always the need to develop a truly solid adjective algorithm. Some (Garver, 2002; Lammert 2002) made strides, but not enough work has been accomplished. While that future programmer will need to think of what case frames will be necessary (I suggest *equiv-equiv, member-class,* and *object-property*), what I believe is more important is to first think about adjectives conceptually. As a word class, they are more complex then at first glance—how do we explain “big” in “big elephant” versus “big mosquito”, for instance. Try to think of adjectives in relation to an algorithm in a top-down fashion. For the interested, I suggest a good starting place to be Connor Ferris’ *The Meaning of Syntax: A Study in the Adjectives of English*, who provides a good insight into adjectives philosophically as well as syntactically.

Appendix A: Tatterdemalion Demo Run

```
* (demo "tatterdemalion.demo")

File /home/nsigrad/nps1/tatterdemalion.demo is now the source of input.


 CPU time : 0.05

*  ; =======================================================================
; FILENAME:     tatterdemalion.demo
; DATE:         DATE
; PROGRAMMER:   Nicholas P. Schwartzmyer



; Lines beginning with a semi-colon are comments.
; Lines beginning with "^" are Lisp commands.
; All other lines are SNePS commands.
;
;
;                  IMPORTANT
; =========================================================
; To use this file: due to the use of SNeBR in this demo
; please run SNePS2.61:
;
;       :ld /projects/shapiro/Sneps/sneps2.61
;
;       types (sneps), and at the SNePS prompt (*), type:
;
;       (demo "tatterdemalion.fullBK.demo" :av)
;
; Make sure all necessary files are in the current working directory
; or else use full path names.
; =======================================================================

; Turn off inference tracing.
; This is optional; if tracing is desired, then delete this.
 ^(
--> setq snip:*infertrace* nil)
nil



 CPU time : 0.00

*
; Load the appropriate definition algorithm:
;;  No accepted adjective algorithm used on this demo
;;


; Load SNeBRIO for the derived contradiction in this demo:
  ^(
--> load "/projects/shapiro/Sneps/new-snebrio")
;   Fast loading /projects/shapiro/Sneps/new-snebrio.fasl
```

```
Warning: read-contr-h-option, :operator was defined in
         /projects/snwiz/sneps-04/Sneps/snebr/sniphandler.lisp and is
         now being defined in /projects/shapiro/Sneps/new-snebrio.cl
Warning: read-sneps-option, :operator was defined in
         /projects/snwiz/sneps-04/Sneps/snebr/snepshandler.lisp and is
         now being defined in /projects/shapiro/Sneps/new-snebrio.cl
Warning: offer-choice, :operator was defined in
         /projects/snwiz/sneps-04/Sneps/snebr/snepshandler.lisp and is
         now being defined in /projects/shapiro/Sneps/new-snebrio.cl
Warning: delete-which, :operator was defined in
         /projects/snwiz/sneps-04/Sneps/snebr/snepshandler.lisp and is
         now being defined in /projects/shapiro/Sneps/new-snebrio.cl
Warning: enquire-hyp-fate, :operator was defined in
         /projects/snwiz/sneps-04/Sneps/snebr/snepshandler.lisp and is
         now being defined in /projects/shapiro/Sneps/new-snebrio.cl
Warning: check-removed, :operator was defined in
         /projects/snwiz/sneps-04/Sneps/snebr/snepshandler.lisp and is
         now being defined in /projects/shapiro/Sneps/new-snebrio.cl
Warning: user-says-yes-1, :operator was defined in
         /projects/snwiz/sneps-04/Sneps/snebr/snepshandler.lisp and is
         now being defined in /projects/shapiro/Sneps/new-snebrio.cl
t



 CPU time : 0.03

*

; Clear the SNePS network:
(resetnet t)

Net reset


 CPU time : 0.01

*
; OPTIONAL:
; UNCOMMENT THE FOLLOWING CODE TO TURN FULL FORWARD INFERENCING ON:
;
; ;enter the "snip" package:
;^(in-package snip)
;
; ;turn on full forward inferencing:
;^(defun broadcast-one-report (represent)
;   (let (anysent)
;     (do.chset (ch *OUTGOING-CHANNELS* anysent)
;        (when (isopen.ch ch)
;         (setq anysent
;              (or (try-to-send-report represent ch)
;                  anysent)))))
;    nil)
;
; ;re-enter the "sneps" package:
; ^(in-package sneps)
; load all pre-defined relations:
(intext "/projects/rapaport/CVA/STN2/demos/rels")
File /projects/rapaport/CVA/STN2/demos/rels is now the source of input.


 CPU time : 0.00

*

(a1 a2 a3 a4 after agent against antonym associated before cause class
 direction equiv etime event from in indobj instr into lex location
 manner member mode object on onto part place possessor proper-name
 property rel skf sp-rel stime subclass superclass subset superset
 synonym time to whole kn_cat)

 CPU time : 0.04

*

End of file /projects/rapaport/CVA/STN2/demos/rels


 CPU time : 0.04

*

; define relations necessary for this passage, but absent from the CVA case
frame dictionary:
(define mod head obj1 obj2 equiv skolem-function)
 equiv is already defined.

(mod head obj1 obj2 equiv skolem-function)

 CPU time : 0.00

*
; load all pre-defined path definitions:
(intext "/projects/rapaport/CVA/mkb3.CVA/paths/paths")
File /projects/rapaport/CVA/mkb3.CVA/paths/paths is now the source of input.


 CPU time : 0.00

*
before implied by the path (compose before
                            (kstar (compose after- ! before)))
before- implied by the path (compose (kstar (compose before- ! after))
                             before-)


 CPU time : 0.00

*
after implied by the path (compose after
                           (kstar (compose before- ! after)))
after- implied by the path (compose (kstar (compose after- ! before))
                            after-)
 CPU time : 0.01

*
sub1 implied by the path (compose object1- superclass- ! subclass
                          superclass- ! subclass)
sub1- implied by the path (compose subclass- ! superclass subclass- !
                           superclass object1)


 CPU time : 0.00

*
super1 implied by the path (compose superclass subclass- ! superclass
                            object1- ! object2)
super1- implied by the path (compose object2- ! object1 superclass- !
                             subclass superclass-)


 CPU time : 0.00

*
superclass implied by the path (or superclass super1)
superclass- implied by the path (or superclass- super1-)


 CPU time : 0.00

*

End of file /projects/rapaport/CVA/mkb3.CVA/paths/paths


 CPU time : 0.02

*


; THE PASSAGE
; ==========================================================

;  "Trains go almost everywhere, and tickets cost
;   roughly two dollars an hour for  first-class travel
;   (first-class Romanian-style that is, with tatterdemalion
;   but comfortably upholstered compartments
;   and equally tatterdemalion but solicitous attendants.)"


; Tayler, J. (1997), "Transylvania Today", The Atlantic Monthly 279(6): 50-
54;
; http://www.theatlantic.com/issues/97jun/transyl.htm
; ===========================================================



; BACKGROUND KNOWLEDGE:
```

```
; =====================

;PRAGMATIC BACKGROUND KNOWLEDGE
;==============================


; Geo-Political
; =============

;; There is an object that is a country

(describe (assert member #Romania class (build lex "country")))

(m2! (class (m1 (lex country))) (member b1))

(m2!)

 CPU time : 0.04

*




;; This country has the name Romania

(describe (assert object *Romania proper-name (build lex "Romania")))

(m4! (object b1) (proper-name (m3 (lex Romania))))

(m4!)

 CPU time : 0.00

*




;; It also has the property of being poor

(describe (assert object *Romania property (build lex "impoverished")))

(m6! (object b1) (property (m5 (lex impoverished))))

(m6!)

 CPU time : 0.01

*
```

```
;; Rule: Any head modified by "Romanian" inherits properties of Romania

(describe (assert forall ($a $y $x)
                                   &ant
                       ((build member *x class
                        (build mod (build lex "Romanian") head *a))
                                     (build object *Romania property *y))
                                   cq
                                    (build object *x property *y)))

(m8! (forall v3 v2 v1)
 (&ant (p3 (object b1) (property v2))
  (p2 (class (p1 (head v1) (mod (m7 (lex Romanian))))) (member v3)))
 (cq (p4 (object v3) (property v2))))

(m8!)

 CPU time : 0.01

*



;;  Rule: If a class is modified by Romanian,
;;  then Romanian Class is a subset of Class (whatever that may be),
therefore
;;  (presumably) if x is a member of the class Romanian Class,
;;  then it will be a member of the class, Class


;(describe (assert forall (*a *x $mod)
                ;ant (build member *x class (build mod *mod head *a))
                 ;cq (build member *x class *a)))




; Infrastructure rules
; ====================



;; Rule: Countries have infrastructure

(describe (assert forall $country
                ant
                (build member *country class (build lex "country"))
                cq
                (build object
                        (build skolem-function
                        infrastructure\ of possessor *country)
                       rel (build lex "infrastructure")
                       possessor *country)))
```

```
(m10! (forall v4) (ant (p5 (class (m1 (lex country))) (member v4)))
 (cq
  (p7 (object (p6 (possessor v4) (skolem-function infrastructure of)))
   (possessor v4) (rel (m9 (lex infrastructure))))))

(m10!)

 CPU time : 0.01

*
```

```
;; Rule: Traits of nations are transitive with respect to their
infrastructure

(describe (assert forall (*country *y)
                &ant
                ((build member *country class (build lex "country"))
                (build object *country property *y))
                cq
                (build object
                        (build skolem-function
                        infrastructure\ of  possessor *country)
                        property *y)))
(m11! (forall v4 v2)
 (&ant (p8 (object v4) (property v2))
  (p5 (class (m1 (lex country))) (member v4)))
 (cq
  (p9 (object (p6 (possessor v4) (skolem-function infrastructure of)))
   (property v2))))

(m11!)

 CPU time : 0.00

*
```

```
;; Rule: Trains are infrastructure

(describe (assert forall $t
        ant
        (build member *t class (build lex "train"))
        cq
        (build member *t class (build lex "infrastructure"))))

(m13! (forall v5) (ant (p10 (class (m12 (lex train))) (member v5)))
 (cq (p11 (class (m9 (lex infrastructure))) (member v5))))

(m13!)

 CPU time : 0.00

*
```

```
;; Rule for member-class transitivity:
;; Trains inherit the properties of infrastructure

(describe (assert forall (*country *t *y)
        &ant
          ((build object
                (build skolem-function infrastructure\ of possessor *country)
                property *y)
                (build member *t class (build lex "train")))
        cq
        (build object *t property *y)))

(m14! (forall v5 v4 v2)
 (&ant (p10 (class (m12 (lex train))) (member v5))
  (p9 (object (p6 (possessor v4) (skolem-function infrastructure of)))
   (property v2)))
 (cq (p12 (object v5) (property v2))))

(m14!)

 CPU time : 0.01

*
```

```
;; Rule: Parts of trains inherit properties of trains

(describe (assert forall (*t *y $possessed)
        &ant
          ((build member *t class (build lex "train"))
          (build object1 *t
                 rel (build lex "have")
                 object2 *possessed)
          (build object *t property *y))
        cq
         (build object *possessed property *y)))
```

```
(m16! (forall v6 v5 v2)
 (&ant (p13 (object1 v5) (object2 v6) (rel (m15 (lex have))))
  (p12 (object v5) (property v2))
  (p10 (class (m12 (lex train))) (member v5)))
 (cq (p14 (object v6) (property v2))))

(m16!)

 CPU time : 0.01

*




;; Rule: To a good degree, the properties of a train's
;; parts reflect on the properties of the train as a
;;  whole

(describe (assert forall (*t *y *possessed)
        &ant ((build member *t class (build lex "train"))
                (build object1 *t
                        rel (build lex "have")
                        object2 *possessed)
                (build object *possessed property *y))
        cq (build object *t property *y)))

(m17! (forall v6 v5 v2)
 (&ant (p14 (object v6) (property v2))
  (p13 (object1 v5) (object2 v6) (rel (m15 (lex have))))
  (p10 (class (m12 (lex train))) (member v5)))
 (cq (p12 (object v5) (property v2))))

(m17!)

 CPU time : 0.01

*





; First-class rules
; =================


;;  Anything that is First class train travel
;;  is a member of the more general class of first-class travel


(describe (assert forall $vartrvl
        ant
        (build member *vartrvl class
                       (build mod
                          (build mod (build lex "first\ class")
                                 head (build lex "train"))
```

```
                              head (build lex "travel")))
        cq
        (build member *vartrvl class
                        (build mod (build lex "first\ class")
                               head (build lex "travel")))))

(m23! (forall v7)
 (ant
  (p15
   (class
    (m21 (head (m20 (lex travel)))
     (mod (m19 (head (m12 (lex train)))
           (mod (m18 (lex first class)))))))
   (member v7)))
 (cq (p16 (class (m22 (head (m20)) (mod (m18)))) (member v7))))

(m23!)

 CPU time : 0.01

*



;; And from the above, all first class travel is a subclass of travel

(describe (assert  forall *vartrvl
        ant
        (build member *vartrvl class
                (build mod (build lex "first\ class")
                       head (build lex "travel"))= fctravel)
        cq
        (build member *vartrvl class (build lex "travel"))))

(m24! (forall v7)
 (ant
  (p16
   (class
    (m22 (head (m20 (lex travel))) (mod (m18 (lex first class)))))
   (member v7)))
 (cq (p17 (class (m20)) (member v7))))

(m24!)

 CPU time : 0.00

*



;; This is meant to be general rule that says that forall things,
;; if they are a member of a class train travel that is doubly
;; modified, i.e. [Romanian-style [First Class [train travel]]]
;; then it is also simply train travel.


(describe (assert forall (*mod * head *vartrvl)
        ant
        (build member *vartrvl class
                        (build mod
                                (build mod
                                (build mod *mod head *head)
                                       head (build lex "train"))
                        head (build lex "travel")))
         cq
        (build member *vartrvl class
                        (build mod
                                (build lex "train")
                                head
                                (build lex "travel")))))

(m28! (forall v7)
 (ant
  (p18
   (class
    (m26 (head (m20 (lex travel)))
     (mod (m25 (head (m12 (lex train)))))))
   (member v7)))
 (cq (p19 (class (m27 (head (m20)) (mod (m12)))) (member v7))))

(m28!)

 CPU time : 0.00

*

;; Rule: First class travel is expensive.
;; More precisely, tickets for first class travel are expensive


(describe (assert forall $tk
        &ant
        ((build member *tk class (build lex "tickets"))
        (build object1 *tk
               rel (build lex "for")
               object2 *fctravel))
        cq
        (build object *tk property (build lex "expensive"))))

(m32! (forall v8)
 (&ant
  (p21 (object1 v8)
   (object2
    (m22 (head (m20 (lex travel))) (mod (m18 (lex first class)))))
   (rel (m30 (lex for))))
  (p20 (class (m29 (lex tickets))) (member v8)))
 (cq (p22 (object v8) (property (m31 (lex expensive))))))

(m32!)

 CPU time : 0.01

*
```

```
;; First class travel has particluar properties,
;; it is usually comfortable and of high quality, to name a few


(describe (assert forall *vartrvl
        ant
        (build member *vartrvl class *fctravel)
        cq
        (build object *vartrvl property
                                (build lex "comfortable"))
         cq
        (build object *vartrvl property
                                (build lex "of\ high\ quality"))))

(m35! (forall v7)
 (ant
  (p16
   (class
    (m22 (head (m20 (lex travel))) (mod (m18 (lex first class)))))
   (member v7)))
 (cq (p24 (object v7) (property (m34 (lex of high quality))))
  (p23 (object v7) (property (m33 (lex comfortable))))))

(m35!)

 CPU time : 0.00

*


;; If tickets for first class travel cost two dollars
;; (such as they do in this passage),
;; then they are not expensive, which means
;;  it is not (actually) first class


(describe (assert forall (*vartrvl *tk)
        &ant
                ((build member *tk class (build lex "tickets"))
                (build object (build lex "two\ dollars")
                        rel (build lex "cost")
                        possessor *tk)
                (build object1 *tk
                        rel (build lex "for")
                        object2 *vartrvl))
        cq
                (build min 0 max 0
                arg
                        (build object *tk
                                property (build lex "expensive"))
                arg
                        (build member *vartrvl class *fctravel))))

(m38! (forall v8 v7)
 (&ant (p26 (object1 v8) (object2 v7) (rel (m30 (lex for))))
  (p25 (object (m36 (lex two dollars))) (possessor v8)
   (rel (m37 (lex cost))))
  (p20 (class (m29 (lex tickets))) (member v8)))
 (cq
  (p27 (min 0) (max 0)
   (arg (p22 (object v8) (property (m31 (lex expensive))))
    (p16
     (class
      (m22 (head (m20 (lex travel))) (mod (m18 (lex first class)))))
     (member v7))))))

(m38!)

 CPU time : 0.01

*




;;  So following from the above, is it likely not the case
;;  that the travel will be so welcoming.
;;  We will unfortunately eschew some of the modality
;;  of a NL and opt for strict negation

(describe (assert forall *vartrvl
        &ant
        ((build member *vartrvl class (build lex "travel"))
        (build object *vartrvl property (build lex "high\ quality")))
        cq
        (build member *vartrvl class *fctravel)))

(m40! (forall v7)
 (&ant (p28 (object v7) (property (m39 (lex high quality))))
  (p17 (class (m20 (lex travel))) (member v7)))
 (cq
  (p16 (class (m22 (head (m20)) (mod (m18 (lex first class)))))
   (member v7))))

(m40!)

 CPU time : 0.01

*






; Properties of Properties
; =======================

;; To use with but rule: comfortable is a positive modifier

(describe (assert
               member (build lex "comfortable")
```

```
                class (build lex "positive attributes")))

(m42! (class (m41 (lex positive attributes)))
 (member (m33 (lex comfortable))))

(m42!)

 CPU time : 0.00

*

;; As above solicitious is a positive modifier

(describe (assert
                member (build lex "solicitous")
                class (build lex "positive attributes")))

(m44! (class (m41 (lex positive attributes)))
 (member (m43 (lex solicitous))))

(m44!)

 CPU time : 0.00

*


; BUT rule: context specific version
; ==================================


; TODO: Replace with more general rule/rules to capture
; the semantics of "but"

; This version basically states that if two attributes opposed
; by "but" and the second is a member of the class positive attributes,
; then the first will be a member of the class of negative attributes
; This positive/negative distinction seems to capture one
; basic use of "but"...
; This rule also goes on to equate this member of class negative attributes
; with the very general property "negative quality".
; Following the CVA group protocol and my reading of the passage,
; I do not think anything of a truly more explicit nature
; can be gleaned about the meaning of "tatterdemalion"

; This needs to be modified as it is too specific: it is not
; the case that the first will always be negative in a 'but'
; relationship, it was merely true for this passage.


(describe (assert forall ($attribute1 $attribute2)
                &ant ((build object1 *attribute1
                             rel (build lex "but")
                             object2 *attribute2)
                       (build member *attribute2
                              class (build lex "positive attributes")))
                cq (build member *attribute1
                                class (build lex "negative attributes"))
                cq
                   (ant (build object *entity property *attribute1)
                    cq (build object *entity property (build lex "negative
quality")))
                cq (build equiv *attribute1 equiv (build lex "negative
quality"))))

(m49! (forall v10 v9)
 (&ant (p30 (class (m41 (lex positive attributes))) (member v10))
  (p29 (object1 v9) (object2 v10) (rel (m45 (lex but)))))
 (cq (p33 (equiv (m47 (lex negative quality)) v9))
  (m48 (property (m47))) cq (p32 (property v9)) ant
  (p31 (class (m46 (lex negative attributes))) (member v9))))

(m49!)

 CPU time : 0.02

*





;               The Linguistic/Pragmatic Unification Rule
; ================================================================

;;  The following rule is my way to join together the pragmatic context, i.e.
;;  knowledge about Romania (well, actually not so much: that's for the
future;
;;  perhaps a rule that states if something is called first class travel in
;;  an impoverished nation, then its probably not first class travel)
;;  about infastructure, and about first class travel with the other context,
;;  the linguistic, namely the "but" construction.


;;  What the rule says is this, if something is not first class travel,
;;  which we come to believe about RSFC train travel after we revise
;;  our beliefs, and if there is a train that has qualities in a but
;;  relationship with each other, then the first of these is will
;;  mean second rate.

;;  This rule should be an immediate future change. For one,
;;  while the logic mirrors that of the protocols, in the rule,
;;  there is a certain disconnect in the logic--the train ant is
;;  not really linked to the negation of member-class ant,
;;  the train ant should incorporate that these are the trains
;;  used for this not first class travel, and probably that notions
;;  of property inheritance between nations and their trains.

;;  Also see above comments about my usage of the 'but' relationship


(describe (assert forall (*possessed *vartrvl *attribute1 *attribute2 *t)
        &ant
```

```
          ((build min 0 max 0
                   arg
                (build member *vartrvl class *fctravel))
           ;(build member *vartrvl class
                ;                      (build mod (build lex "train")
                                       ;       head (build lex "travel")))
           (build object1 *t
                  rel (build lex "have")
                  object2 *possessed)
           (build object *possessed property *attribute1)
           (build object *possessed property *attribute2)
           (build object1 *attribute1
                  rel (build lex "but")
                  object2 *attribute2))
        cq
           (build equiv *attribute1 equiv (build lex "second rate")))))
(m51! (forall v10 v9 v7 v6 v5)
 (&ant (p36 (object v6) (property v10)) (p35 (object v6) (property v9))
  (p34 (min 0) (max 0)
   (arg
    (p16
     (class
      (m22 (head (m20 (lex travel))) (mod (m18 (lex first class)))))
     (member v7))))
  (p29 (object1 v9) (object2 v10) (rel (m45 (lex but))))
  (p13 (object1 v5) (object2 v6) (rel (m15 (lex have)))))
 (cq (p37 (equiv (m50 (lex second rate)) v9))))

(m51!)

 CPU time : 0.01

*
Warning: ignoring extra right parenthesis on #<echo-stream>






; CASSIE READS THE PASSAGE:
; =========================



;Trains of Romanian provenance go almost everywhere in Transylvania

(describe (add agent
                (build  mod (build  lex "Romanian")
                        head (build lex "train"))
               act
                (build action (build lex "go\ everywhere\ in")
                       object (build lex "Transylvania"))))

(m56!
 (act (m55 (action (m53 (lex go everywhere in)))
       (object (m54 (lex Transylvania)))))
 (agent (m52 (head (m12 (lex train))) (mod (m7 (lex Romanian))))))

(m56!)

 CPU time : 0.05

*



; Romanian trains are trains

(describe (add member
                (build mod (build lex "Romanian")
                       head (build lex "train"))
                 class (build lex "train")))

(m68! (object (m52 (head (m12 (lex train))) (mod (m7 (lex Romanian)))))
 (property (m5 (lex impoverished))))
(m67! (object (m66 (possessor b1) (skolem-function infrastructure of)))
 (property (m5)))
(m58! (class (m9 (lex infrastructure))) (member (m52)))
(m57! (class (m12)) (member (m52)))
(m6! (object b1) (property (m5)))
(m2! (class (m1 (lex country))) (member b1))

(m68! m67! m58! m57! m6! m2!)

 CPU time : 0.58

*







;Romanian-style first-class trains are a subclass

(describe (add member
                (build mod (build mod
                                (build lex "Romanian-style")
                                  head
                                (build lex "first\ class"))
                      head (build lex "train") = rsfctrain)
                class
                  (build mod (build lex "Romanian")
                         head (build lex "train"))))

(m77! (class (m52 (head (m12 (lex train))) (mod (m7 (lex Romanian)))))
 (member
    (m76 (head (m12))
     (mod (m75 (head (m18 (lex first class)))
           (mod (m74 (lex Romanian-style))))))))
```

```
(m77!)

 CPU time : 0.02

*



;;  Romanian-Style first class train travel
;; are members of the class of first class travel

(describe (add member
                (build mod
                      (build mod
                        (build mod (build lex "Romanian-style")
                                head (build lex "first\ class"))
                              head (build lex "train"))
                         head (build lex "travel"))
               class
                *fctravel))

(m82!
 (object
  (m78 (head (m20 (lex travel)))
   (mod (m76 (head (m12 (lex train)))
         (mod (m75 (head (m18 (lex first class)))
               (mod (m74 (lex Romanian-style)))))))))
 (property (m33 (lex comfortable))))
(m81! (object (m78)) (property (m34 (lex of high quality))))
(m80! (class (m20)) (member (m78)))
(m79! (class (m22 (head (m20)) (mod (m18)))) (member (m78)))

(m82! m81! m80! m79!)

 CPU time : 0.03

*





;; Contradiction resolution
;; This was supossed to rid the demo of the need for SNeBR,
;; but I've yet to get it working.

;;   (describe (assert
;;               &ant ((build member
;;                       (build mod
;;                               (build mod
;;                                       (build mod (build lex "Romanian-
style")
;;                                              head (build lex "first\
class"))
;;                                        head (build lex "train"))
;;                               head (build lex "travel"))
;;                            class *fctravel)
;;            (build min 0 max 0 arg
;;             (build member
;;                       (build mod
;;                               (build mod
;;                                       (build mod (build lex "Romanian-
style")
;;                                              head (build lex "first\
class"))
;;                                        head (build lex "train"))
;;                               head (build lex "travel"))
;;                    class *fctravel)))
;;       cq (build min 0 max 0 arg
;;               (build member
;;                       (build mod
;;                               (build mod
;;                                       (build mod (build lex "Romanian-
style")
;;                                              head (build lex "first\
class"))
;;                                        head (build lex "train"))
;;                               head (build lex "travel"))
;;                       class *fctravel))))




;tickets for RSFC travel exist

(describe (add
                object1 #ticket
                rel (build lex "for")
                object2 (build mod
                                (build mod
                                        (build mod (build lex "Romanian-
style")
                                               head (build lex "first\
class"))
                                        head (build lex "train"))
                               head (build lex "travel"))))

(m83! (object1 b2)
 (object2
  (m78 (head (m20 (lex travel)))
   (mod (m76 (head (m12 (lex train)))
         (mod (m75 (head (m18 (lex first class)))
               (mod (m74 (lex Romanian-style))))))))))
 (rel (m30 (lex for))))

(m83!)

 CPU time : 0.01

*


; An object call ticket is a member of the class of tickets
```

```
(describe (add member *ticket class (build lex "tickets")))

(m84! (class (m29 (lex tickets))) (member b2))

(m84!)

 CPU time : 0.01

*


;; =========================================================
;; IMPORTANT

;; The following rule will derive the contradiction that will
;; invoke SNeBR. While it is actually in the hands of the
;; demo-runner as to which hypothesis to discard,
;; the hypothesis that is intended to be discarded is #4,
;; which states that RSFC travel is first class travel.

;; Demo-runner: please enter following commands at SNeBR
;; prompt:
;; --Type 'r' to restart the run
;; --Type 4 to select this hypothesis
;; --Type 'd' to delete it from the set
;; --Type 'q' to quit revising the set
;; --Inspecting the other hypotheses is optional, but do not
;;   delete any
;; --Type 'no' when asked if it is desired
;;   to add a new hypothesis

;; ==========================================================
;; MAKE SURE YOU READ THE ABOVE BEFORE HITTING RETURN!!!!!

;  These tickets cost two dollars,
;  literally, "the tickets have the cost of two dollars"
;  cost is a stative verb, not an act,
;  thus I am violating the SNePS rule of thumb
;  which uses agent-act-action-object for verbs.
;  My approach may be novel, but I think, using
;  only preestablished CVA case frames,
;  it most adequately captures the meaning.
;  Certain states in languages in fact must be represent
;  by possession, take Spanish,
;  "Tengo vienticinco anos" or "Tiene frio"


(describe (add object
                (build lex "two\ dollars")
                rel
                (build lex "cost")
                possessor
                *ticket))

  A contradiction was detected within context default-defaultct.
  The contradiction involves the newly derived proposition:
    (m89! (min 0) (max 0)
 (arg
  (m79!
   (class
    (m22 (head (m20 (lex (travel)))) (mod (m18 (lex (first class))))))
   (member
      (m78 (head (m20 (lex (travel))))
       (mod (m76 (head (m12 (lex (train))))
             (mod (m75 (head (m18 (lex (first class))))
                   (mod (m74 (lex (Romanian-style)))))))))))))
  and the previously existing proposition:
    (m79!
 (class
  (m22 (head (m20 (lex (travel)))) (mod (m18 (lex (first class))))))
 (member
    (m78 (head (m20 (lex (travel))))
     (mod (m76 (head (m12 (lex (train))))
           (mod (m75 (head (m18 (lex (first class))))
                 (mod (m74 (lex (Romanian-style))))))))))
  You have the following options:
   1. [C]ontinue anyway, knowing that a contradiction is derivable;
   2. [R]e-start the exact same run in a different context which is
      not inconsistent;
   3. [D]rop the run altogether.

  (please type c, r or d)
=><= r


  In order to make the context consistent you must delete at least
  one hypothesis from each of the following sets of hypotheses:
    (m85! m84! m83! m79! m38!)
    (m85! m84! m83! m79! m38!)

   The hypotheses listed below are included in more than
   one set. Removing one of these will make more than one
   set consistent.
    (m85! m84! m83! m79! m38!)


 In order to make the context consistent you must delete
  at least one hypothesis from the set listed below.

An inconsistent set of hypotheses:

 1 : (m85! (object (m36 (lex (two dollars)))) (possessor (b2))
 (rel (m37 (lex (cost)))))          (4 supported propositions: (m89! m88!
m87! m85!) )

 2 : (m84! (class (m29 (lex (tickets)))) (member (b2)))          (4 supported
propositions: (m89! m88! m87! m84!) )

 3 : (m83! (object1 (b2))
 (object2
  (m78 (head (m20 (lex (travel))))
   (mod (m76 (head (m12 (lex (train))))
         (mod (m75 (head (m18 (lex (first class))))
               (mod (m74 (lex (Romanian-style)))))))))
 (rel (m30 (lex (for)))))          (4 supported propositions: (m89! m88! m87!
m83!) )

 4 : (m79!
 (class
  (m22 (head (m20 (lex (travel)))) (mod (m18 (lex (first class))))))
 (member
    (m78 (head (m20 (lex (travel))))
     (mod (m76 (head (m12 (lex (train))))
           (mod (m75 (head (m18 (lex (first class))))
                 (mod (m74 (lex (Romanian-style))))))))))          (4
supported propositions: (m82! m81! m80! m79!) )

 5 : (m38! (forall v8 v7)
 (&ant (p26 (object1 v8) (object2 v7) (rel (m30 (lex (for)))))
  (p25 (object (m36 (lex (two dollars)))) (possessor v8)
   (rel (m37 (lex (cost)))))
  (p20 (class (m29 (lex (tickets)))) (member v8)))
 (cq
  (p27 (min 0) (max 0)
   (arg (p22 (object v8) (property (m31 (lex (expensive)))))
    (p16
     (class
      (m22 (head (m20 (lex (travel))))
       (mod (m18 (lex (first class))))))
     (member v7))))))          (4 supported propositions: (m89! m88! m87!
m38!) )


  Enter the list number of a hypothesis to examine or
  [d] to discard some hypothesis from this list,
  [a] to see ALL the hypotheses in the full context,
  [r] to see what you have already removed,
  [q] to quit revising this set, or
  [i] for instructions

  (please type a number OR d, a, r, q or i)
=><= 4

  (m79!
 (class
  (m22 (head (m20 (lex (travel)))) (mod (m18 (lex (first class))))))
 (member
    (m78 (head (m20 (lex (travel))))
     (mod (m76 (head (m12 (lex (train))))
           (mod (m75 (head (m18 (lex (first class))))
                 (mod (m74 (lex (Romanian-style))))))))))
 WFFS that depend on m79!:

  (m82!
 (object
  (m78 (head (m20 (lex (travel))))
   (mod (m76 (head (m12 (lex (train))))
         (mod (m75 (head (m18 (lex (first class))))
               (mod (m74 (lex (Romanian-style)))))))))
 (property (m33 (lex (comfortable)))))
```

```
  (m81!
 (object
  (m78 (head (m20 (lex (travel))))
   (mod (m76 (head (m12 (lex (train))))
         (mod (m75 (head (m18 (lex (first class))))
               (mod (m74 (lex (Romanian-style))))))))))
 (property (m34 (lex (of high quality)))))
  (m80! (class (m20 (lex (travel))))
 (member
    (m78 (head (m20 (lex (travel))))
     (mod (m76 (head (m12 (lex (train))))
           (mod (m75 (head (m18 (lex (first class))))
                 (mod (m74 (lex (Romanian-style)))))))))))
  (m79!
 (class
  (m22 (head (m20 (lex (travel)))) (mod (m18 (lex (first class))))))
 (member
    (m78 (head (m20 (lex (travel))))
     (mod (m76 (head (m12 (lex (train))))
           (mod (m75 (head (m18 (lex (first class))))
                 (mod (m74 (lex (Romanian-style)))))))))))

  What do you want to do with hypothesis m79!?
  [d]iscard from the context, [k]eep in the context,
  [u]ndecided, [q]uit revising this set, [i]nstructions
  (please type d, k, u, q or i)
=><= d

The consistent set of hypotheses:

 1 : (m85! (object (m36 (lex (two dollars)))) (possessor (b2))
 (rel (m37 (lex (cost)))))          (4 supported propositions: (m89! m88!
m87! m85!) )

 2 : (m84! (class (m29 (lex (tickets)))) (member (b2)))          (4 supported
propositions: (m89! m88! m87! m84!) )

 3 : (m83! (object1 (b2))
 (object2
  (m78 (head (m20 (lex (travel))))
   (mod (m76 (head (m12 (lex (train))))
         (mod (m75 (head (m18 (lex (first class))))
               (mod (m74 (lex (Romanian-style))))))))))
 (rel (m30 (lex (for)))))          (4 supported propositions: (m89! m88! m87!
m83!) )

 4 : (m38! (forall v8 v7)
 (&ant (p26 (object1 v8) (object2 v7) (rel (m30 (lex (for)))))
  (p25 (object (m36 (lex (two dollars)))) (possessor v8)
   (rel (m37 (lex (cost)))))
  (p20 (class (m29 (lex (tickets)))) (member v8)))
 (cq
  (p27 (min 0) (max 0)
   (arg (p22 (object v8) (property (m31 (lex (expensive)))))
    (p16
     (class
      (m22 (head (m20 (lex (travel))))
       (mod (m18 (lex (first class))))))
     (member v7))))))         (4 supported propositions: (m89! m88! m87!
m38!) )


  Enter the list number of a hypothesis to examine or
  [d] to discard some hypothesis from this list,
  [a] to see ALL the hypotheses in the full context,
  [r] to see what you have already removed,
  [q] to quit revising this set, or
  [i] for instructions

  (please type a number OR d, a, r, q or i)
=><= q

  The following (not known to be inconsistent) set of
  hypotheses was also part of the context where the
  contradiction was derived:
    (m77! m57! m56! m51! m49! m44! m42! m40! m35! m32! m28! m24! m23! m17!
m16! m14! m13! m11! m10! m8! m6! m4! m2!)

  Do you want to inspect or discard some of them?
=><= no

  Do you want to add a new hypothesis?
=><= no

(m92! (min 0) (max 0) (arg (m91 (property (m31 (lex expensive))))))
(m89! (min 0) (max 0)
 (arg
  (m79
   (class
    (m22 (head (m20 (lex travel))) (mod (m18 (lex first class)))))
   (member
      (m78 (head (m20))
       (mod (m76 (head (m12 (lex train)))
             (mod (m75 (head (m18))
                   (mod (m74 (lex Romanian-style)))))))))))
(m88! (min 0) (max 0) (arg (m86 (object b2) (property (m31)))))
(m87! (min 0) (max 0) (arg (m86) (m79)))
(m85! (object (m36 (lex two dollars))) (possessor b2)
 (rel (m37 (lex cost))))
(m84! (class (m29 (lex tickets))) (member b2))
(m83! (object1 b2) (object2 (m78)) (rel (m30 (lex for))))
(m77! (class (m52 (head (m12)) (mod (m7 (lex Romanian)))))
 (member (m76)))
(m68! (object (m52)) (property (m5 (lex impoverished))))
(m67! (object (m66 (possessor b1) (skolem-function infrastructure of)))
 (property (m5)))
(m57! (class (m12)) (member (m52)))
(m6! (object b1) (property (m5)))
(m2! (class (m1 (lex country))) (member b1))

(m92! (min 0) (max 0) (arg (m91 (property (m31 (lex expensive))))))
(m89! (min 0) (max 0)
 (arg
  (m79
   (class
    (m22 (head (m20 (lex travel))) (mod (m18 (lex first class)))))
   (member
      (m78 (head (m20))
       (mod (m76 (head (m12 (lex train)))
             (mod (m75 (head (m18))
                   (mod (m74 (lex Romanian-style))))))))))))
(m88! (min 0) (max 0) (arg (m86 (object b2) (property (m31)))))
(m87! (min 0) (max 0) (arg (m86) (m79)))
(m85! (object (m36 (lex two dollars))) (possessor b2)
 (rel (m37 (lex cost))))
(m84! (class (m29 (lex tickets))) (member b2))
(m83! (object1 b2) (object2 (m78)) (rel (m30 (lex for))))
(m77! (class (m52 (head (m12)) (mod (m7 (lex Romanian)))))
 (member (m76)))
(m68! (object (m52)) (property (m5 (lex impoverished))))
(m67! (object (m66 (possessor b1) (skolem-function infrastructure of)))
 (property (m5)))
(m57! (class (m12)) (member (m52)))
(m6! (object b1) (property (m5)))
(m2! (class (m1 (lex country))) (member b1))

(m92! m89! m88! m87! m85! m84! m83! m77! m68! m67! m57! m6! m2!)

 CPU time : 0.98

*
```

```
;RSFC trains have compartments

(describe (add  object1 *rsfctrain
                rel (build lex "have")
                object2 #compartments))

(m112! (object1 (m12 (lex train))) (object2 b3) (rel (m15 (lex have))))

(m112!)

 CPU time : 0.03

*

(describe (add member *compartments class (build lex "compartments")))

(m114! (class (m113 (lex compartments))) (member b3))

(m114!)

 CPU time : 0.01
```

```
*







;RSFC trains have attendants

(describe (add  object1 *rsfctrain
                rel (build lex "have")
                object2 #attendants))

(m115! (object1 (m12 (lex train))) (object2 b4) (rel (m15 (lex have))))

(m115!)

 CPU time : 0.02

*

(describe (add member *attendants class (build lex "attendants")))

(m117! (class (m116 (lex attendants))) (member b4))

(m117!)

 CPU time : 0.01

*




;These compartments have the quality of being tatterdemalion

(describe (add object *compartments
                property (build lex "tatterdemalion")))

(m120! (property (m118 (lex tatterdemalion))))
(m119! (object b3) (property (m118)))
(m112! (object1 (m12 (lex train))) (object2 b3) (rel (m15 (lex have))))

(m120! m119! m112!)

 CPU time : 0.27

*
```

```
;These compartments are comfortable

(describe (add object *compartments
                property (build lex "comfortable")))

(m132! (property (m33 (lex comfortable))))
(m122! (object b3) (property (m33)))

(m132! m122!)

 CPU time : 0.03

*
```

```
;  Comfortable/Solicitous and tatterdemalion are in a
;  "but" relationship with each other


(describe (add object1 (build lex "tatterdemalion")
                rel (build lex "but")
                object2 (build lex "comfortable")))

(m134! (equiv (m118 (lex tatterdemalion)) (m50 (lex second rate))))
(m133! (object1 (m118)) (object2 (m33 (lex comfortable)))
 (rel (m45 (lex but))))

(m134! m133!)

 CPU time : 0.06

*


(describe (add object1 (build lex "tatterdemalion")
                rel (build lex "but")
                object2 (build lex "solicitous")))

(m135! (object1 (m118 (lex tatterdemalion)))
 (object2 (m43 (lex solicitous))) (rel (m45 (lex but))))

(m135!)

 CPU time : 0.02

*
```

```
;The attendants are tatterdemalion
```

```
(describe (add object *attendants
                property (build lex "tatterdemalion")))

(m136! (object b4) (property (m118 (lex tatterdemalion))))
(m120! (property (m118)))
(m115! (object1 (m12 (lex train))) (object2 b4) (rel (m15 (lex have))))

(m136! m120! m115!)

 CPU time : 0.31

*
```

```
;These attendants are solicitous

(describe (add object *attendants
                property (build lex "solicitous")))

(m149! (property (m43 (lex solicitous))))
(m148! (object b4) (property (m43)))
(m134! (equiv (m118 (lex tatterdemalion)) (m50 (lex second rate))))

(m149! m148! m134!)

 CPU time : 0.06

*
```

```
; Ask Cassie what "tatterdemalion " means:
;==========================================



;                        Equivalences
; =====================================================



; Following our "but" rule, tatterdemalion should be equivalent
; to a negative quality:

(describe (deduce
                equiv (build lex "tatterdemalion")
                equiv (build lex "negative quality")))

(m150! (equiv (m118 (lex tatterdemalion)) (m47 (lex negative quality))))
```

```
(m150!)

 CPU time : 0.02

*

; Cassie agrees!




; Following our unified linguistic/pragmatic rule,
; the meaning of "tatterdemalion" should be something like
; 'second rate':

(describe (deduce
                equiv (build lex "tatterdemalion")
                equiv (build lex "second rate")))

(m134! (equiv (m118 (lex tatterdemalion)) (m50 (lex second rate))))

(m134!)

 CPU time : 0.00

*

; Cassie agrees!



;                       Class Membership
; ===============================================================

; Properties can be members of larger class of properties, e.g.
; the property 'green' is a member of the class of properties
;  we refer to as 'colors'

;  Following the protocols, the most we can say about "tatterdemalion"
;  is that it is (probably) a negative attribute, let us see
;  if Cassie agrees, following the rules given to her:

(deduce member (build lex "tatterdemalion") class (build lex "negative
attributes"))

(m151!)

 CPU time : 0.02

*

;  She does.
;                        Modified Entities
;  ===============================================================

; Lets find the nodes with object- property arc with "tatterdemalion":

(dump (find (object- property) (build lex "tatterdemalion")))

(b4
 (member- (m147 m146 m144 m143 m141 p124 m117!) object-
  (m148! p122 m140 m139 m138 m137 p121 p120 m136!) object1-
  (p126 m142 p125) object2- (p127 p123 m115!) possessor- (m145)))
(b3
 (member- (m131 m130 m128 m127 m125 p116 m114!) object-
  (p112 m124 m123 m122! m121 p111 p110 m119!) object1- (p118 m126 p117)
  object2- (p119 p115 m112!) possessor- (m129)))

(b4 b3)

 CPU time : 0.00

*


; And what are these entities?

(describe m114 m117)
(m117! (class (m116 (lex attendants))) (member b4))
(m114! (class (m113 (lex compartments))) (member b3))

(m117! m114!)

 CPU time : 0.00

*

End of /home/nsigrad/nps1/tatterdemalion.demo demonstration.
```

Appendix B: Syntax and Semantics of Case Frames Used

The following “standard” CVA case frames were

act-act-action-object

object1-rel-object2

member-class

object-property

object-rel-possessor

lex

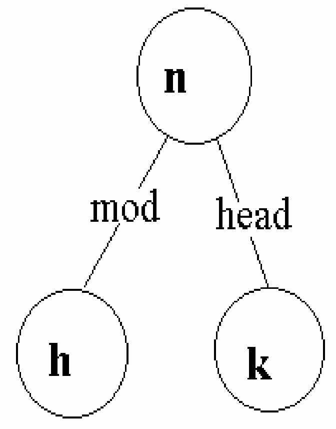
object-proper-name

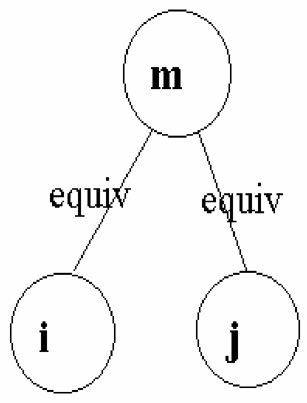
Syntax and semantics of the above are available at
http://www.cse.buffalo.edu/~rapaport/CVA/CaseFrames/case-frames/

In addition the SNePS negation case frame was needed:

min-0 max 0-arg

Its syntax and semantics can be found in the SNePS case frame dictionary:
http://www.cse.buffalo.edu/sneps/ Bibliography/bibliography.html#Manuals





Semantics

[[n]] is the proposition that [[h]] modifies [[k]]

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

Ferris, Connor. (1993), *The Meaning of Syntax: A Study in the Adjectives of English* (New York: Longman).

Garver, Christopher. (2002), “Adjective Representation in Contextual Vocabulary Acquisition”, http://www.cse.buffalo.edu/~rapaport/CVA/cvapapers.html.

Goldfain, Albert. (2003), “Computationally Defining “Harbinger” via Contextual Vocabulary Acquisition”, http://www.cse.buffalo.edu/~rapaport/CVA/cvapapers.html.

Hastings, Peter M. & Lytinen, Steven L. (1994), "Objects, Actions, Nouns, and Verbs" *Proceedings of the 16th Annual Conference of the Cognitive Science Society* (Hillsdale, NJ: Lawrence Erlbaum Associates): 397-402.

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

Kibby, Michael; Rapaport, William; Morgan Brain; Santosh ?; Xi, Yulei; Wieland, Karen; Robey, Wendy. CVA Group Think-Aloud “Schmalion” (nonword equivalent for “tatterdemalion”) Midwinter 2002/2003. http://www.cse.buffalo.edu/~rapaport/CVA/tatterdemalion-protocol.html

Lammert, Adam. (2002). “Defining Adjectives Through Contextual Vocabulary Acquisition”, http://www.cse.buffalo.edu/~rapaport/CVA/cvapapers.html.

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

Rapaport, William J., & Ehrlich, Karen (2000), "A Computational Theory of Vocabulary Acquisition" in Lucja M. Iwanska & Stuart C. Shapiro (eds.), *Natural Language Processing and Knowledge Representation: Language for Knowledge and Knowledge for Language* (Menlo Park, CA/Cambridge, MA: AAAI Press 347-375.

Rapaport, William J., & Kibby, Michael W. (2002), "ROLE: Contextual Vocabulary Acquisition: From Algorithm to Curriculum".

Segal, Erwin M., & Duchan, Judith F. (1997), “Interclausal Connectives as Indicators of Structuring in Narrative”, in Jean Costermans & Michel Fayol (eds.), *Processing Interclausal Relationships: Studies in the Production and Comprehension of Text* (Mahwah, NJ: Lawrence Erlbaum Associates): 95-119.

Shapiro, Stuart C., & Rapaport, William J. (1987), "SNePS Considered as a Fully Intensional Propositional Semantic Network", in Nick Cercone & Gordon McCalla (eds.), *The Knowledge Frontier: Essays in the Representation of Knowledge* (New York: Springer-Verlag): 262-315.

Shapiro, Stuart C. & Rapaport, William J. (1995), "An Introduction to a Computational Reader of Narrative", in Judith Felson Duchan, Gail A. Bruder, & Lynne E. Hewitt (eds.) Deixis in Narrative: A Cognitive Science Perspective (Hillsdale, NJ: Lawrence Erlbaum Associates): 79-105.

Sternberg, Robert J.; Powell, Janet S.; & 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.

Tayler, J. (1997), "Transylvania Today", *The Atlantic Monthly* 279(6): 50-54. http://www.theatlantic.com/issues/97jun/transyl.htm

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

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