Speaking of Elephants: Generating from Knowledge Representations

Scott D. Anderson

COINS Technical Report 89-19

Abstract

The knowledge bases employed by AI programs ought to be usable both for driving inferential reasoning and for generating natural language text. Consequently, this paper looks at the problem of generating natural language text from knowledge bases using a standard AI knowledge representation. The research has emphasized the generation of complex noun phrases because AI knowledge bases tend to be about *things*. The extra demands of natural language have motivated some changes to the form of the knowledge representation but the resulting KR formalism still seems to be useful for general AI reasoning. The approach addresses, but does not solve, problems with quantification, predication, order of adjectives, and relative clauses.

I have demonstrated the approach with a small program that takes expressions using an example knowledge base and builds an input message for the natural language generator Mumble-86, which then generates text. The program and the knowledge base are presented in the appendices. I have left to the reader's intuition the claim that this knowledge representation discipline is still useful for AI reasoning.

This work, which was originally the author's Master's project, was supported in part by the Air Force Systems Command, Rome Air Development Center, Griffiss AFB, New York, 13441 under contract No. F30602-85-C-0008. This contract supports the Northeast Artificial Intelligence Consortium (NAIC).

Contents

List of Figures

Chapter 1

Realization

AI programs, almost by definition [11], reason from explicit representations of knowledge. Many of these programs would also like to talk, to communicate their knowledge and their conclusions to their users. Therefore, the Generator (the natural language generation component of an integrated AI system) must be able produce text from expressions couched in terms of the program's knowledge base. By using the program's knowledge representation formalism in a disciplined way, and by annotating it with linguistic information to be used by the Generator, the Generator's job can be made much easier. The KR discipline and annotation should not intrude on the other responsibilities of the KR. Ideally, the KR discipline and annotation can easily be done by the knowledge engineer, and thus the competence of the Generator will increase smoothly and in parallel with the increasing knowledge of the AI program. Thus, a program will always be able to say what it knows.

1 Mapping Concepts to Words

Programs know about lots of different kinds of concepts, and these concepts are the intensions of open class words such as nouns, verbs and adjectives. Therefore, the most obvious annotation of a knowledge representation is a mapping from concepts to words. Given concepts such as **Elephant**, **eat** and **red** in a KR, we can map these onto the noun "elephant," the verb "eat" and the adjective "red."

However, for some concepts the mapping will not be so clear. For instance, our KB may contain information about elephants that like people, such as the fragment of a structured-inheritance network (SI-Net) [1] shown in figure 1.1.

There is no English word for (the concept of) an elephant that likes people, though there may be a word for it in another language. Call the mapping from concepts to words the word-mapping. This example clearly shows that the word-mapping is at best a *partial* function and that the concepts for which it is defined is language-dependent (as well as its range).

Is the word-mapping even a function? Consider the words "elephant" and "pachyderm." There does not seem to be any semantic difference between the two—the difference is probably one of "register" or "style." You might use "pachyderm" if you want to be pedantic, or you might choose between them for alliterative purposes ("elegant elephant" or "ponderous pachyderm"). There are also slang words such as "tusker." Still, we would want to have the default value of

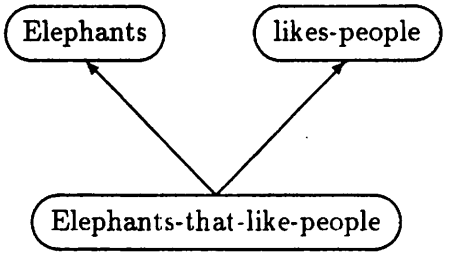

Figure 1.1: A concept for which there is no single English word

the mapping be the common word "elephant."

The word-mapping could certainly be implemented as a complicated function, with each value being a single word but with a multiplicity of inputs to direct the lexical choice: inputs corresponding to register, style, desired phonetic form, and many others. However, I think it is better to divide the work in the following way. The word-mapping will take concepts as its only input and produce either a single word or a special-purpose function. That second function is responsible for choosing between various possible words and can take whatever inputs it finds relevant to the task. Lexical choice, then, is the composition of word-mapping with the set of special-purpose functions. Dividing lexical choice this way allows us to face the difficult problems of lexical choice only when we want to, retaining the ability to use a default in other cases.

Having reduced the word-mapping to a many-to-one (partial) function, we can implement it simply by annotating each concept for which it is defined with the value of the function on that input. Thus, the concept **eat** might be annotated with the verb "eat." The generator is responsible for conjugating the verb properly, modifying it to express tense and aspect, and a host of other problems. The concept **Gray** may simply be annotated with the word "gray."[1] And the concept **Elephants** might be annotated with a function to choose between "elephant" and "pachyderm."

Note that the word-mapping is a many-to-one function because of polysemy. For instance, if we have concepts for **river-bank** and **fiduciary-bank**, they will both map to the word "bank." We could, of course, interprete the word-mapping as a one-to-one function to *senses* of words and have a many-to-one function from word senses to words, but this requires an independent representation of word senses without gaining us anything.

2 Roles as Dimensions

We have cast the first phase of lexical choice as strictly a problem of conceptualization. That is, if an individual is conceptualized as an instance of the concept **Foo**, then we immediately

[1]The fact that knowledge engineers justifiably use mnemonic names makes this look trivial, but remember that the name of concept is a gensym as far as the program is concerned.

have a word for it by evaluating the word-mapping on that input.[2] However, we may have the option of being more specific and hence more informative. Consider the fragment of an SI-Net shown in figure 1.2.

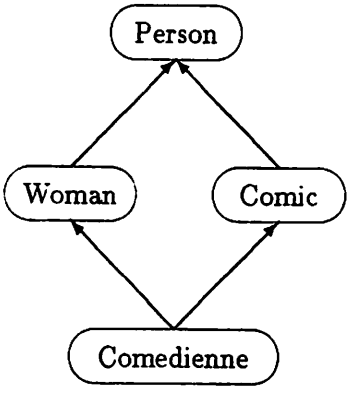

Figure 1.2: Concepts, with associated words, of varing specificity

If we have an object `#S(PERSON NAME "Chris" SEX FEMALE OCCUPATION COMEDY)` that we know is an instance of **Person**, we could simply map it to "person." For instance, we might say "Chris is a person." This is rather dull, but easy. Suppose, however, that our knowledge representation understands that the concept **Person** has the role **Occupation** and that the concept **Comic** means that the value of the role is restricted to the concept **Comedy**, as in the SI-Net shown in figure 1.3.

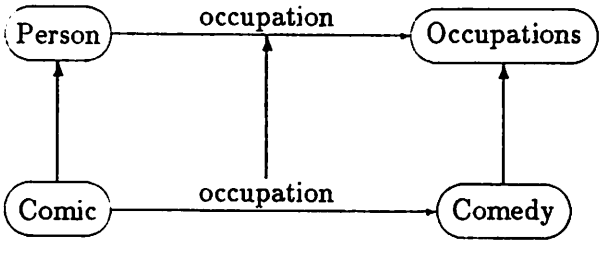

Figure 1.3: A 'comic' is a person whose occupation is 'comedy.'

This network then understands the relationship between these two words, and the Generator can choose to search down from **Person** for a concept that fills the **Occupation** role in the correct way. If the word-mapping is defined for that concept (that is, the language has a word to express that concept), then the Generator can use it.

Similarly, if the generator wants to express Chris's sex, the network can make that possible by representing the semantic relationship of **Person** and **Woman**, as in figure 1.4.

[2] Assuming that the word-mapping is defined for that concept. Later we will address the problem of expressing a concept for which the word-mapping is not defined.

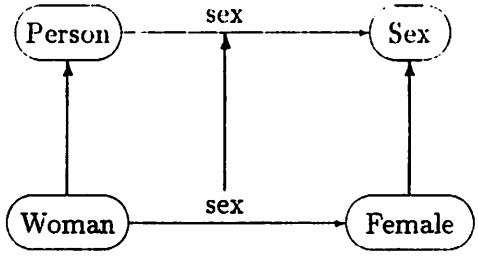

Figure 1.4: A 'woman' is a person whose sex is 'female.'

We can view the *roles* of **Person** as *dimensions* in a search space. The points in the search space are conceptualizations of **Person** that restrict the value of one or more roles, and therefore, if there is a word for that concept, we can express several aspects of the person at once by using that word.

This is all contingent on the individual (Chris, in this case) being an instance of a more abstract concept than necessary. Some knowledge representations may force individuals to be instances of the most specific possible concept. In such a KR, we can still search for other possible words, but we would be searching *upwards* in an effort to *avoid* expressing certain role-restrictions. For instance, many people nowadays will refer to someone as a "person" rather than using "man" or "woman," even when that information is known.

We would have to be more careful in searching upwards, though, because we do not want to accidentally end up saying "hominid" or something even more vague. This is especially problematic because of the Gricean maxim of informativeness [5], which lets our audience infer that, since a more specific term was available but not used, the speaker does not believe the more specific term to be true. Hence, to call someone a "hominid" has the conversational implicature that they are not human, even though they could be. This problem is probably a small one, since the search would presumably terminate at **Person** unless, we indeed wanted to avoid expressing the fact that the individual is a person.

Fortunately, not all knowledge representations will force us to search upwards. For instance, KRYPTON [2] makes a distinction between the *taxonomic* component (which would include our SI-Net concepts and the associated words) and the *assertional* component (where individuals such as Chris would exist). Given that division, we can certainly make Chris an instance of **Person** if we wish, and reserve further classification as we see fit. Rosch's work on basic catagories [9][8] supports the idea of classifying an individual at a more abstract level than necessary, such as **Person** instead of **Comedienne**.

Finally, note that this search space is quite large, because **Person** and other natural kinds can have so many roles and role-values. For instance, **Person** could have at least the following roles:

sex	race	height
weight	nationality	eye color
hair color	occupation	hobbies
age	Zodiacal sign	marital status
children	siblings	

I do not think that the search space is so large that the search problem becomes difficult, but the size does lead to the following observations:

- The Generator will have to decide what it will and will not search for. It may know that someone has a hobby of stamp-collecting, but decide not to mention that information, perhaps because it is not relevant to the conversation. In another conversation, it may search and discover the word "philatelist."

- The knowledge base will have to observe what I call the "Rhombus Rule."[3] Look again at figure 1.2. If the generator searches first for a way to express the **sex** role and next searches for a way to express the **occupation** role, it ends up at **comedienne**. If it orders the searches in the opposite way, it ends at the same place. The roles really are orthogonal dimensions in this search space, and it should not matter which is taken first. Therefore, all of these links are necessary. In many cases, the knowledge engineer will simply have to be disciplined and careful in constructing the knowledge base, but some knowledge representations may have automated aids for this: the classifier used in NIKL ought to discover and add any missing links in a situation like this, but I discovered during programming that the NIKL classifier fails on this rule.

- From an implementational viewpoint, the search space should not be completely laid out, but remain implicit, because many of the possible subconcepts will not have words associated with them. For instance, English has words for women that express their hair color, such as "blonde," "brunette," and "redhead,"[4] but has no word for a person that also expresses their eye color. Therefore storage should not be allocated to represent those useless concepts.[5]

Viewing roles as dimensions partially answers a question posed by Brachman, Fikes, and Levesque [2]. Given a semantic net like that shown in figure 1.5, they asked "how many kinds of rocks are there?" Counting just the subconcepts of **rock**, we would say there are five, which is ridiculous. However, if we distinguish three roles of **rock**, say size, color, and mode of origin, we could count in any of three different directions. In that figure, there is one kind of rock looking at color, one kind of rock looking at size, and three kinds of rock looking at mode of origin.

Having discussed how concepts can be expressed by mapping them to words, we can now restate part of the Generator's problem as follows: Given the word-mapping, which is a partial function, we must devise a *total* function which can take any concept in the knowledge base and express it in English. In many cases, there will be an English word that expresses the concept and the word-mapping is sufficient. In other cases, the concept will have to be expressed based on its formal relationship to other concepts in the knowledge base, ones for which the language *does* have words. Because the input to many natural language generation systems,

[3] If we move to higher dimensions, we'll see that the Rhombus Rule probably ought to be called the Hypercube rule, since we always want to end up at the same corner of the Hypercube, regardless of the order that we traverse the edges.

[4] It is curious that English has no word expressing black hair.

[5] The usefulness of concepts is discussed in chapter 2, page 14.

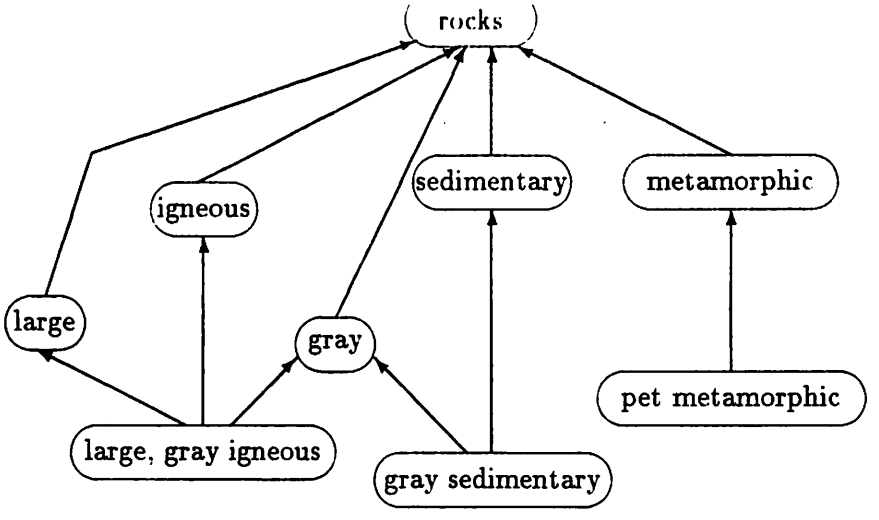

Figure 1.5: A semantic net of rocks

such as Mumble-86, is called the *message*, we will call the total function that maps concepts to messages the message-mapping.

Chapter 2

Common Nouns

The first approach to describing something for which there is no word was suggested by our discussion of the person Chris, above. That is, we can search for a more specific, or failing that, a more general term. Suppose that Clyde is an elephant in the circus, using the SI-Net shown in figure 2.1.

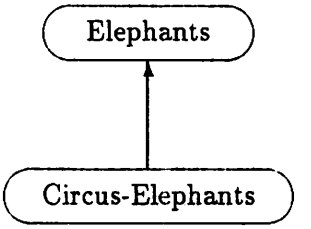

Figure 2.1: Failing a word for Circus Elephants, we can use one for Elephants.

If we have no word for **Circus-Elephants**, we can simply go up the superconcept cable and say that Clyde is an Elephant. Would that it were so easy. If Clyde were an old elephant, we would build the network shown in figure 2.2.

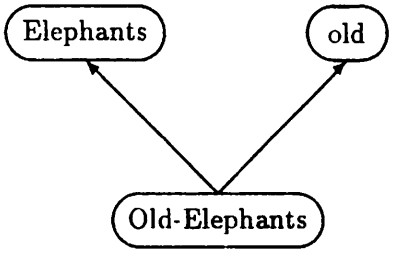

Figure 2.2: We could describe old elephants as elephants but not as 'olds.'

Apparently, we can either say that Clyde is an elephant or an old, whatever that means.[1] Distinguishing between nouns and verbs in the range of the word-mapping is not sufficient to tell the Generator what to do, since many adjectives may also be used as nouns. (In fact, in French, one can say *mon vieux*, which translates literally as "my old." This may, however, be idiomatic, since I don't think that every old thing can be referred to as *un(e) vieux*.) What we really need is the notion of *Sort*.

1 Distinguishing Sorts from other Unary Predicates

In general, we can draw two parallel distinctions: "elephant" is a noun, while "old" is an adjective, and **Elephant** is a *Sort* while **old** is a property. Sorts are the intensions of common nouns and are semantically and logically different from properties, even though both can be expressed as unary predicates; sorts have inhabited the literature on philosophy and semantics since at least 1965.[2] All unary predicates have a Principle of Application, which says for what entities in a given model structure the predicate is true; Sorts have in addition a Principle of Identity. Furthermore, sorts can combine with quantifiers, which predicates cannot.

1.1 The Principle of Identity

The Principle of Application determines the extension of a predicate, that is, it says what things in a particular world fall under that predicate, for instance, what things are elephants and what are old or large. The Principle of Identity considers correspondences between things in different worlds. Essentially, it considers the question, "is this the same K." Clearly, this only makes sense for sorts: we can ask whether this is the same elephant as that, but not the same "old" or the same "large."

Gupta elaborates the differences between the Principles of Identity and Application [6, pp 20–26], and I will only briefly summarize them here. Gupta sees three distinct principles associated with Sorts:

1. The Principle of Persistence
2. The Principle of Trans-World Identity
3. The Principle of Application

The first two essentially comprise the Principle of Identity and distinguish sorts from predicates.

The Principle of Persistence traces objects through time. For instance, is this river the same river as an hour ago, or a year ago? This principle can be either simple or complex, depending on philosophy. For people, it can be simple: I prefer to think of myself as the same person as I have been throughout my life. But there have been many arguments about when a particular collection of planks can be called the 'same ship' as another collection at a different time.

---

[1] Ignore for the moment the optimal description, which is to combine the two and say that Clyde is an old elephant.

[2] However, the precious little that I know about sortal logic comes from [6], and so that book is a reference for this entire chapter.

The Principle of Trans-World Identity is complicated. Would I still be me if I were born at a different place? A different time? Named differently? Brought up differently? All of the above? Combining this principle with the Principle of Persistence to yield the Principle of Identity allows us to say when d_1 in world w_1 at time t_1 is the same as d_2 in world w_2 at time t_2.

1.2 Quantification

Sorts also differ from predicates in whether they can combine with quantifiers. We can say "Every elephant is ..." but not "Every large is ...." Since only sorts can combine with quantifiers, we change the syntax of quantification to the following:

$$[Q\ K, x]\ \alpha$$

Q is a quantifier, such as $\forall$ or $\exists$, K is a sort, x is a variable and α is a formula. Thus, we could have the following rule, which states that all elephants are gray:

$$[\forall \text{ Elephant},x]\ \text{gray}(x)$$

The sortal quantification that Gupta proposes is not equivalent to the usual, unrestricted quantification. Actually, unrestricted quantification is a special case of sortal quantification, where there is only one sort—"thing."

One advantage of sortal quantification is in interpreting predicates that are relative to the domain of predication, as with **old**. If we have

$$[\exists \text{ Elephant},x]\ \text{old}(x)$$

which states that some elephants are old, and we know that **old** is relative, we know in this case what it is relative to. After all, old for elephants is different from old for people or mice.

Another advantage of sortal quantification is in allowing other kinds of quantifiers, such as **most**, as can any quantification that has a restrictive clause and a dependent clause. For instance, the truth conditions of

$$[\textit{most}\ \text{Elephant},x]\ \text{nice}(x)$$

are straightforward, and we can easily read it as "most elephants are nice." In an unrestricted quantification, we have *most* x, $\textit{elephant}(x) \supset \textit{nice}(x)$, which has odd truth conditions: if most things are not elephants, then for most values of x the expression $\text{elephant}(x) \supset \text{nice}(x)$ is true (since $a \supset b$ is true when a is false), so the quantification is true, regardless of whether most elephants are nice. Rendering the second quantification into English is also more difficult. Thus, both from the reasoning component's and the Generator's points of view, the sortal quantification is superior to unrestricted quantification.

In summary, there is a semantic difference between sorts and predicates that should be reflected in our knowledge representation. The Generator can use the representational difference to aid it in finding related terms for a concept. Essentially, by allowing only sorts as superconcepts of sorts, we turn the superconcept link into an "a kind of" (AKO) link, and turn the that part of the network into a hierarchy of kinds.

Chapter 3

Quantifiers

How would we represent in a knowledge base a rule such as "most elephants are nice?" Certainly, we cannot make **elephants** a subconcept of **nice** (because **nice** is not a sort) or even **nice-animals**, even though the latter is a sort, because such a representation would be indistinguishable from the statement that *all* elephants are nice.

To represent a rule like this, we must represent the following parts: the quantifier, the sort, the variable, and the dependent clause. Therefore, we might be able to represent rules by using the usual semantic-net trick of breaking up a frame into a bunch of binary predicates. For an example, see figure 3.1.

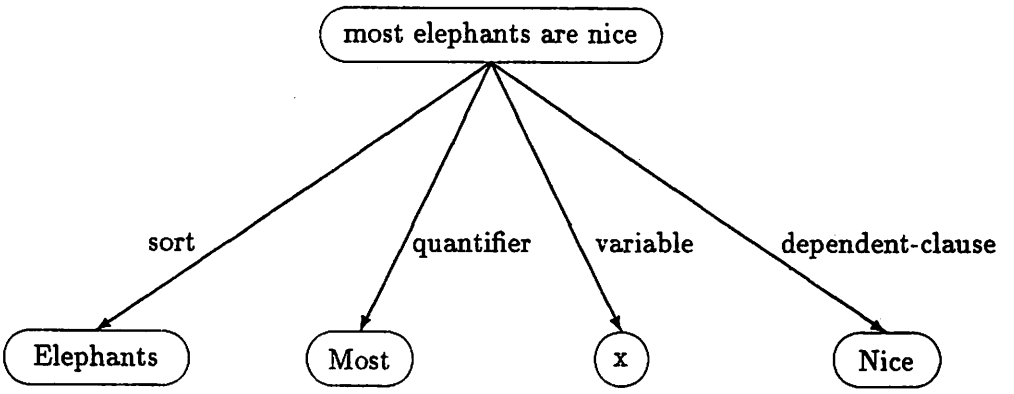

Figure 3.1: A possible semantic net representation of "most elephants are nice."

Of course, the representation of the dependent clause turned out in this case to be simple, but in general we can parse the clause into a tree and put the root of the tree at the head of the **dependent-clause** arc. We can still have inheritance with this representation, simply by declaring that the inverse of the **quantifier** role is an inheritable link. Thus, if we know that circus elephants are a kind of elephant, we can plausibly infer that they may be nice.[1]

[1]Note that this is merely a plausible inference, not a truth-preserving one. It is plausible as long as being a circus elephant is not antithetical with being nice, as being a dangerous elephant would.

An alternative representation is to use the facility of most modern AI KRs to attach (inheritable) data to a concept. That is, we will write down a quantificational rule using PC and then attach that rule to the concept that is quantified over—the sort.

But if we're using PC, why bother with the semantic net at all? Clearly, PC is sufficient for our representational needs, and may even be easier, since we don't have to atomize higher-arity predicates into a bunch of binary ones. That is, we don't have to turn a predicate like **bet(agent,patient,theme1,theme2)** into a network like the one in figure 3.2.

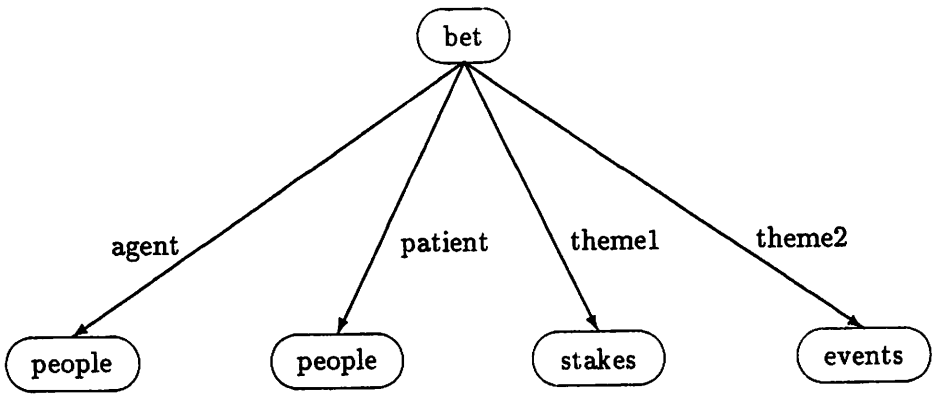

Figure 3.2: The semantic net representation of the four-place predicate "bet."

The reason for having two representations is not notational adequacy, but computational efficiency. Certain important relationships can be computed more quickly from a semantic net than from formulas in PC.[7] Furthermore, we will be *partitioning* our PC formulas by placing them on sorts in a semantic net, so that, for instance, if we are reasoning about elephants and not people, all the rules about elephants will be conveniently accessible, but the ones about people will not be in the way.

1 Attributive Information

Given the scheme of representing quantifications in PC and attaching them to sorts, we can extend the scheme to other quantifiers. Consider the rule that all elephants are gray:

$$[\forall \text{ Elephants},x]\ \text{gray}(x)$$

Since **gray** is not a sort, we do not want it to be a superconcept of **Elephants**, yet we will want to be able to represent this rule. With this scheme, we can simply write the rule using PC and attach it to the sort.

Consider two more rules:

1. [$\forall$ triangle,x]number-of-sides(x) = 3
2. [$\forall$ triangle, x] [$\exists$ angle α_1,α_2,α_3]
   $\text{in}(\alpha_1, x) \wedge \text{in}(\alpha_2, x) \wedge \text{in}(\alpha_3, x) \wedge$
   $\alpha_1 \neq \alpha_2 \wedge \alpha_2 \neq \alpha_3 \wedge \alpha_3 \neq \alpha_1 \wedge$
   sum(value-of(α_1),value-of(α_2),value-of(α_3)= 180°

Clearly, we could represent both rules either in the semantic net or in PC.[2] They are different kinds of information however: the first is definitional while the second is attributive—it is additional information about triangles.[3] Since we always want our programs to know what the definition of something is, distinct from other information about it, we must distinguish these rules. I suggest that the semantic net record only definitional information; for instance, an SI-net can represent the definition of a triangle as a value restriction on the range of the **number-of-sides** role that is inherited from **polygon**. Attributive information, such as the second rule, should be attached to the appropriate sort. Indeed, all the attached rules, whether quantified with **all** or some other quantifier, represent attributive information.

We can look again at the elephant concept in this light. The rule that

$$[\forall \text{ elephant},x]\ \text{mammal}(x)$$

is definitional information that is recorded in the structure of the network, while the rule that

$$[\forall \text{ elephant},x]\ \text{gray}(x)$$

is attributive information, and therefore is stored as attached data to **elephant**.

Even though we are using two representation languages—semantic nets and the predicate calculus—the two are not wholly separate. Since the net records definitional information, it is there that we will look for the definitions of the predicates and other terms used in the PC expressions. This is similar to the A-box/T-box distinction of Krypton [2].

2 Subset Information

The flip side of the rule that all elephants are mammals is that some mammals are elephants; that is:

$$[\forall \text{ elephant}, x]\ \text{mammal}(x)$$

versus

$$[\exists \text{ mammal}, x]\ \text{elephant}(x)$$

(Of course, these rules are not equivalent, since the first implies the second, but the second does not imply the first. To see this, consider the rule that some extinct species were mammals,

[2]Actually, both formulas make use of functions, "number-of-sides" and "sum,"so they're not in first-order predicate calculus, per se, but we have not made a strong commitment to FOPC, merely to some non-network language.

[3]I recall seeing this triangle example and the definitional/attributive distinction in Brachman's work, but I cannot find the reference.

which certainly does not imply that all mammals are extinct species.) We can view the ISA or AKO link between the the two concepts as representing both those rules.

Will all rules of the form

$$[\exists\ K, x]\ \mathrm{L}(x)$$

be represented by an ISA link? Not necessarily. We can, if we want, represent the same information by attaching the quantification directly to the concept **K**. Consider the following two rules: (1) some elephants are circus elephants and (2) some elephants are dangerous. Both circus elephants and dangerous elephants could be represented as a subconcept of elephants, as shown in figure 2.1 (p 7). On the other hand, we could attach a rule to **elephants**, which would obviate having to build the subconcept.

When should a concept be built? What makes a concept useful?

- From the reasoning component's viewpoint, a concept is useful if there are rules that depend on that concept, since a concept serves as the domain of quantificational rules, and thereby is the locus of knowledge about something.
- From the Generator's viewpoint, a concept is useful as the intension of a word (in the same way that a proposition is the intension of a sentence), and thereby supplies the domain of the word-mapping.

Though logically independent, these viewpoints may overlap to a large extent because people may invent words to express concepts that they want to use in reasoning. To go back to an earlier example (see page 5), English has words expressing the hair color of people but not the eye color. The stereotypes we have about blondes and redheads may have motivated the coining and retaining of those words, while "blueye" was never coined because we have no clear stereotypes about people with blue eyes. Thus, **blonde** is useful as both the locus of that stereotype and the intension of the word "blonde." Though we would build a subconcept to serve the needs of either the reasoning component or the generator, we may find that they generally serve both.

For the case of dangerous elephants and circus elephants, there is no word for either concept, so that criterion is irrelevant. For instance, if we knew something about circus elephants, such as "circus elephants normally live a long time," we would need the subconcept as the domain for that rule. Otherwise, we could simply state the existence of these kinds of elephants on the superconcept. Note that there is nothing wrong with redundantly stating the information, nor with building a subconcept that is not strictly necessary. We ought to build every useful concept, but we can also build others.

3 Normal and Prototypical Information

I just said that we could state a rule like "circus elephants normally live a long time" and attach it to **elephants**. Similarly, we could state a rule such as "normal elephants have four legs." But what kinds of rules are these? What kind of a quantifier do words like "normally" and "normal" come from? We can state the truth conditions of quantifiers such as **all** ($\forall$), **some** ($\exists$), and **most**,[4] but can we do so for **normal**? Perhaps we can define **normal** K, for

[4] **Most** is true iff the dependent clause is true for most (more than half) of the individuals specified by the restrictive clause.

some sort K, as some kind of statistically *average* K, but the averaging mechanism undoubtedly will depend on K—defining the average elephant is very different from averaging numbers.

Whether we can rigorously define **normal** as a quantifier or not, it appears to be very useful, since much of the information in the world is default information. For example, we know a lot about normal elephants, but most of that information can be cancelled for particular elephants (and, similarly, for subconcepts of **elephant**). We know that normal elephants have four legs, but Clyde could have had a birth defect. We know that normal elephants are gray, but certainly there are albino elephants. Normal elephants live in either Africa or India, but captive elephants can live anywhere. In fact, there is very little information that can be stated as true for **all** elephants, but a great deal can be stated about **normal** elephants, and the same is probably so of anything except abstractions such as numbers and geometrical objects. Since we want to represent normal and prototypical information (the two seem equivalent), it makes sense to introduce **normal** as a quantifier. Note that by stating the **normal** rules in PC and attaching them to concepts, we do not sacrifice inheritance, since the attached data is inheritable.

In summary, we will represent definitional information in the semantic network, and attributive information as attached quantifications. We have discussed having **all**, **most**, **normal**,[5] and **some** as quantifiers, and we could extend this set to include others, such as **many**, **several**, **half**, and particular numbers, such as **three**. These rules can be used in non-monotonic reasoning, in the following way:

Given:

$$[normal\ K, x]\ p(x)$$
$$K(a)$$

plausibly conclude:

$$p(a)$$

Our confidence in the inference depends on the quantifier. Intuitively, **most** yields more confidence than **many**, and with a quantifier such as **several**, we may want to make the opposite conclusion—assume $\neg p(a)$ unless we know differently. For instance, if we know that **several** senators are women, and a is a senator, then a is probably not a woman. In any event, the use of non-universal quantifiers such as **normal** means that we must employ non-monotonic logic in our reasoning component.

[5]Linguistically, "normal" does not pattern like a quantifier; it maps to an adjective instead of a determiner. I don't think this is a problem with the approach, but it does entail some special case code.

Chapter 4

Adjectives

Common nouns can be restricted by adding adjectives to them, and this stems from a conceptual difference in which additional properties are predicated of some sort: for instance, **little-brown-jugs** is certainly a subconcept of **jugs**. There are three different kinds of adjectives, reflecting three different kinds of predicates:

intersective These predicates can be defined independently of the sort. For instance, we can characterize the color 'pink' (perhaps as a mixture of ranges of wavelengths) without reference to the thing that is pink. Because 'pink' is intersective, we know that a pink elephant is a pink thing.

subsective These predicates can only be defined relative to the sort that they are predicating. For instance, 'small' is a property with depends on the sort: a small elephant is not a small thing.

alienating These predicates do not produce a subconcept when applied to a concept; they produce something else entirely. For instance, a fake gun is not a gun; an alleged communist is not a communist; and a toy elephant is not an elephant.

This chapter explores how the definition of concepts that involve a sort and a predicate might be done.

1 Intersectives

Had we not decided to restrict the superconcepts of sorts to be sorts, we could easily represent pink elephants as in figure 4.1. After all, one reason for the term 'intersective' is that the extension of **pink-elephants** is the interchapter of the extensions of **elephants** and **pink**.

But maybe we've been wrong about the predicate 'pink' all along; maybe that predicate doesn't really exist, and instead we have **color(x,pink)**, represented as in figure 4.2.

This representational scheme has two advantages. First, it makes more information explicit, since the unary predicate **pink** loses or, at best, disguises its relationship to the binary predicate **color**. In this scheme, if we want to have **pink** as a predicate, we can always create it via lambda abstraction: $\lambda\, x\ color(x, pink)$. Second, the increase in explicit information expands the choices available to the generator.

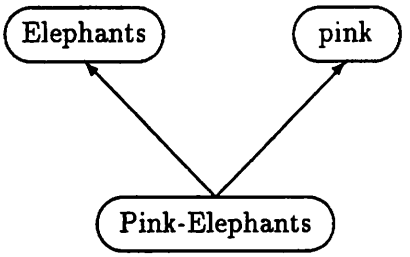

Figure 4.1: A representation for Pink Elephants that we will not use.

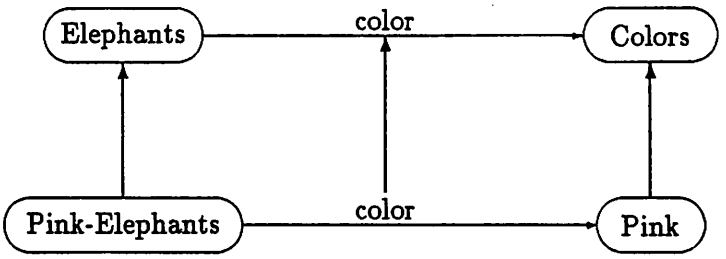

Figure 4.2: A 'pink elephant' is one whose 'color' is 'pink.'

In other words, where we once could say only:

- pink elephants, or
- elephants that are pink

we can now say, in addition:

- pink-colored elephants, and
- elephants whose color is pink

Usually, we will prefer the first choices, as they are more compact, but there may be occasions when the latter are preferable.

2 Subsectives

With subsective predicates, we would not have been tempted to represent them as in figure 4.1, since a predicate such as **small** does not have an extension. For instance, a particular mouse may be large qua mouse, but small qua pet. Thus, an entity cannot be declared to be 'small' until it is categorized in some Sort, such as **mice**, and the measurement of smallness is made relative to the average or to the prototypical representative of that Sort.

We might represent subsective predicates as we do intersectives, as value restrictions on roles. Figure 4.3 gives a simple example.

However, what are we to do with the type of information shown in figure 4.4? Clearly, a

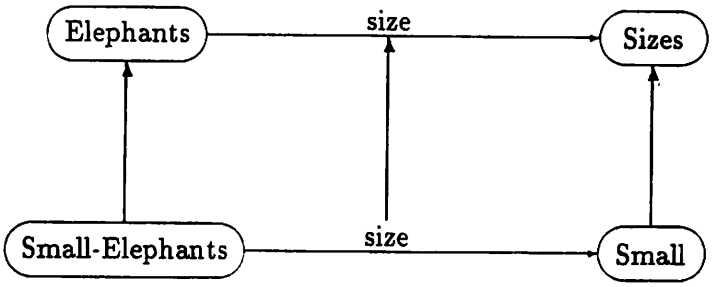

Figure 4.3: 'Small elephants' are those whose 'size' is 'small.'

role such as **size** can range over either quantitative (intersective) or qualitative (subsective) information. Does this mean that predicates are freely mixed together in the knowledge base, as in figure 4.5? This representation seems reasonable.

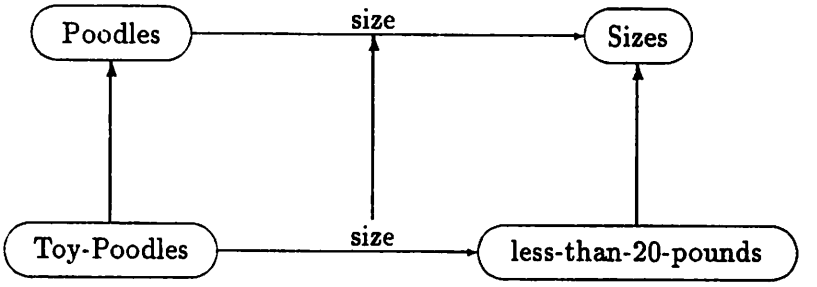

Figure 4.4: Toy poodles are poodles under 20 pounds.

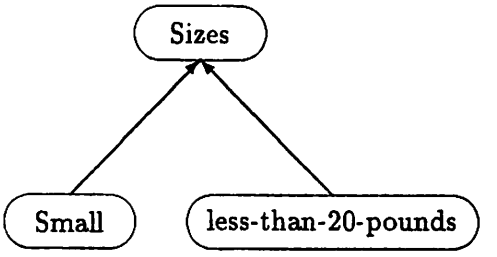

Figure 4.5: A representation in which the predicate **Sizes** has subconcepts which can be either intersective or subsective

Another possibility is to distinguish relative from absolute measures; that is, instead of having a single **size** predicate, we can have **relative-size** and **absolute-size**. This is also reasonable, and may have greater representational power. Consider the intension of 'large toy

poodles,' depicted in figure 4.6.

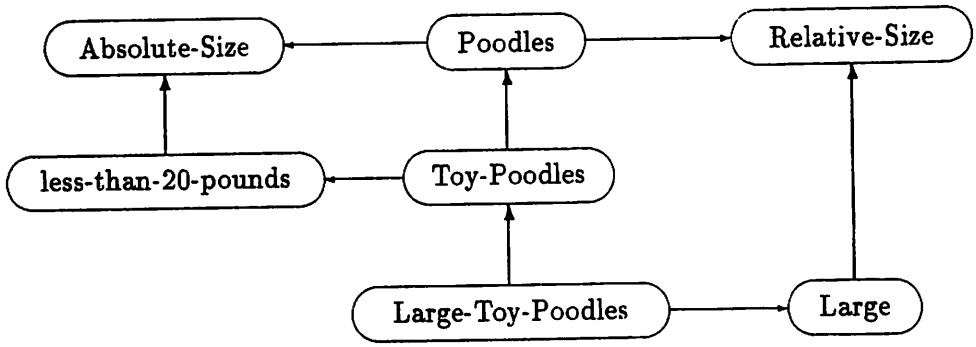

Figure 4.6: Large toy poodles have two roles–absolute-size and relative-size—both of which have been restricted. The role-names have been omitted for typographic sanity.

Another motivation for the dual roles comes from considering the representation of individuals. For instance, we might have an individual elephant, Clyde, who weighs 7,000 pounds and therefore is fat. In the first representational scheme, we would have to store only Clyde's actual weight as the value of his **weight** role, and leave implicit the fact that he is fat, inferring it whenever necessary. But with the second scheme, we could infer Clyde's fatness once, from the value of his **absolute-weight** role, and then store it as the value of his **relative-weight** role.

Distinguishing intersective from subsective adjectives might be useful to the Generator as well, as it seems that subsective adjectives come before intersective adjectives in the NP. For instance, we have:

1a. small pink elephants
1b. * pink small elephants
2a. little brown jug
2b. * brown little jug

However, the needs of the Generator do not choose between the two representations. In the mixed representation (see figure 4.5), we could include this information simply by annotating each predicate with its catagory—intersective or subsective. The second representation is slightly more convenient, as the annotation can be done at a general level and be inherited.

3 Alienating

I have little to say about alienating predicates, except to note their existence as a representational problem. How should a network represent the definition of **toy-elephants**? Conceivably, the concept of **toy** has a **resembles** role, so that the representation of **toy-elephants** is equivalent to the representation of **toys-that-resemble-elephants**—see figure 4.7.

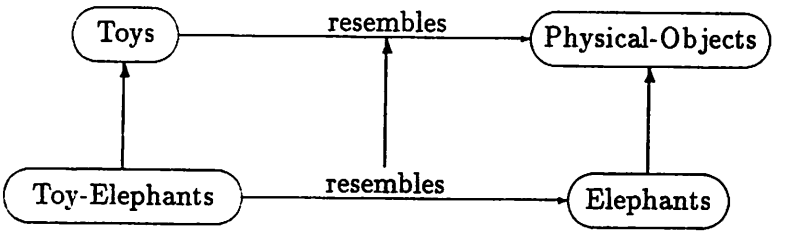

Figure 4.7: Toy elephants are toys that resemble elephants.

Given this representation, the Generator cannot treat **toy** as it does other sorts, at least with respect to modification. In figure 4.1, the Generator can eliminate the role and simply say "pink elephants," but if it did that in figure 4.7, it would say "elephant toys," which is simply wrong. The generator must go to extra effort to come up with the phrasing "toy elephants."

The notion of "toy," like "game," is hard to define precisely. We can think of it as an operator that takes any physical object and makes a toy of it. Cottrell refers to this as the "toyify" operator [3]. This leads to an interesting observation about conceptual differences because of point at which "toyification" occurred, and these conceptual differences are reflected in language. For instance, compare "a toy African elephant" with "an African toy elephant." In the former, "toyification" was applied to an African elephant (as opposed to an Indian elphant), while in the latter, "toyification" was applied to an elephant, and the toy is African (perhaps made there).

Some predicates can be either intersective or alienating. Consider **wooden**, for instance. A wooden table is clearly a kind of table, yet a wooden elephant is not a kind of elephant—it's more like a toy elephant. One way to distinguish these uses is to treat **wooden** as a value restriction on the **material** role. Since **table** has such a role, but **elephant** does not, to predicate **wooden** of an elephant must be alienating.

Since alienating predicates can be so arbitrary, this remains an open area of research in knowledge representation.

4 Number Restriction

Just as **pink** can be reworked as a value restriction on a binary **color** predicate, other intersective predicates can be reworked as binary predicates with number restrictions. For instance, we can take the unary predicate **free**, and turn in into the binary predicate **owner**. Supposing that **elephant** has that role and the number restriction on it is, in KL-One notation, (0,1), which means that the role cannot have fewer than zero fillers nor more than one. A **free** elephant is a subconcept of **elephant** where the number restriction on that role is (0,0), which means individuals in that class cannot be owned. Similarly, a **captive** elephant would be another subconcept of **elephant** with the role's number restriction being (1,1).

The problem with this representation is that, unlike value restriction, number restriction completely loses the predicates **free** and **captive**. There is no first-class object in the knowledge base corresponding to those predicates. This is especially a problem for the generator, since there is no object which can be the intension of open class words such as "free" and "captive."

5 Problematic Adjectives

So far, our representation of unary predicates (adjectives) has been to turn them into arguments of a binary predicate, where the binary predicate is a role of some concept. We may not always be able to do this; this chapter looks at predicates that may pose that problem.

To represent "happy elephants," we can introduce **emotional-state** as a role of elephants, probably with discrete values ranging from **depressed** to **ecstatic**. To represent "sick elephants" or "dying elephants," we might introduce **health** as another role, with values ranging from **fit-as-a-fiddle** through **fine** and right down to **death's-door**. We could also use numeric values, from +10 down to -9, as the early Schank representations did [10].[1] These solutions are completely ad hoc but still plausible.

What about an adjective like "talented?" I cannot come up with a role any more specific than **has-property**. Again, this solution is plausible, but it is also an admission of defeat, since we can just as well name the role **applicable-unary-predicate**.

There is another way that we can admit defeat, and that is to re-introduce unary predicates as super-concepts, but retain the distinction between Sorts and predicates. In figure 4.8, we have indicated Sortal concepts with boxes and unary predicates with ovals. We can maintain the AKO discipline by requiring that any subconcept of a Sort must be a Sort. Essentially, the AKO hierarchy we had before is still there, as a subgraph of the concept graph, found by taking only the nodes that are Sorts (boxes).

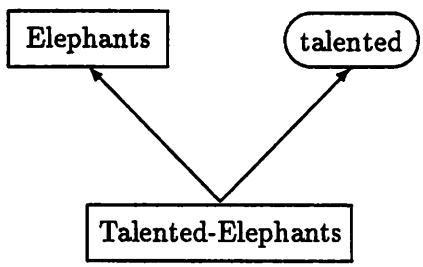

Figure 4.8: A way to admit unary predicates as superconcepts, but retain the hierarchy of Sorts

One advantage of the box scheme is to eliminate ad hoc binary predicates, but the related disadvantage is that we re-admit unary predicates such as **gray**, which really ought to be turned into binary predicates.

[1] I've omitted the value -10, meaning 'dead,' since it seems that 'dead' is like an alienating adjective: is a dead elephant really an elephant anymore?

Chapter 5

Relative Clauses

Another way that noun phrases can be restricted is by relative clauses, and this too can be projected back onto the knowledge representation. We can represent the argument structures of verbs in the usual method of semantic net; indeed, the notion in Linguistics of θ-roles makes this representation quite reasonable. We can restrict Sorts with these verbs plus argument structures to yield concepts that can be expressed as an NP with a restrictive relative clause.

For instance, figure 5.1 shows how we can use the representation for "bet," shown in figure 3.2 (page 12), to define "bettors." The Generator should be able to describe bettors as "people who bet."

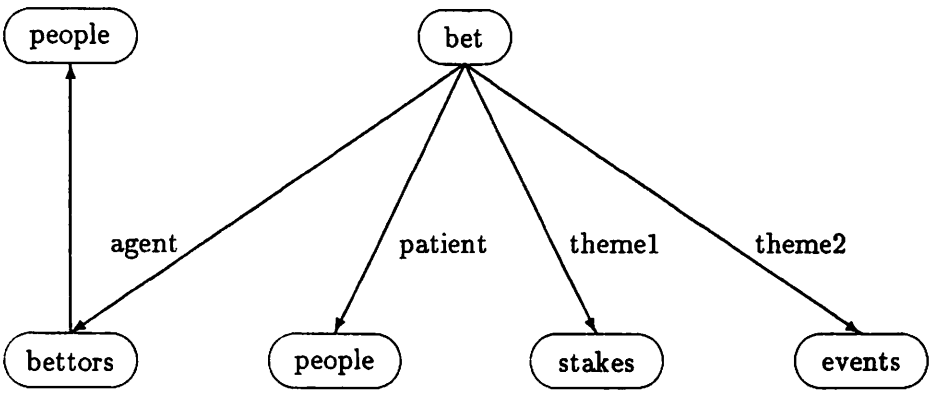

Figure 5.1: Defining 'bettors' as 'people who bet'

The concept being defined need not be the agent of the verb's theta grid. Figure 5.2 shows how both "employer" and "employee" can be defined via the verb "employ."

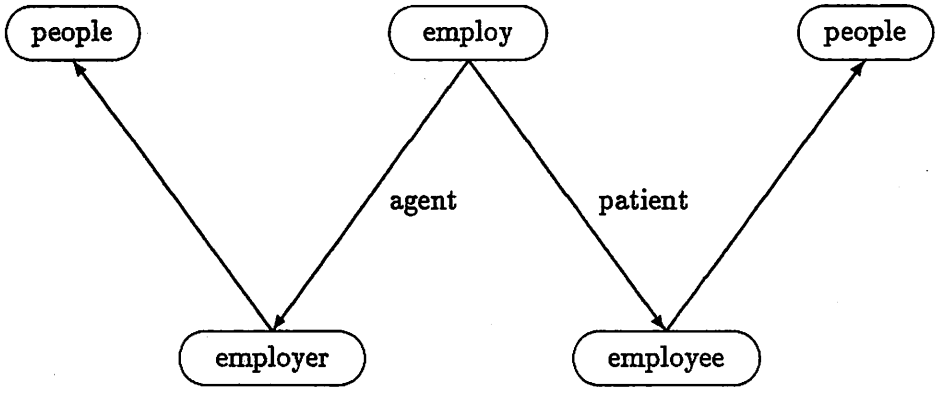

Figure 5.2: Defining 'employer' and 'employee'

Chapter 6

A Problem with This Whole Approach

By projecting the subtle distinctions of language back onto the knowledge representation, we derive a KR which has finely ground the world, perhaps too finely. It may be that we have projected closely related but separate English phrases onto separate conceptual objects, without retaining the close relationship. For example, we have concepts that are the intensions of verbs and concepts which are the intensions of adjectives, but do we connect the concepts, as the words are connected?

Consider the verb "fear" and the adjective "afraid." We might represent the concept "Elephants that fear mice" as shown in figure 6.1, and the concept "Elephants that are afraid of mice" as in figure 6.2. But aren't these two concepts really the same?[1] We let the language motivate a conceptual difference that does not exist.

I believe that the representation of figure 6.1 is probably the correct one, but whichever we pick as the correct representation, the choice yields two problems. First, we leave the Generator responsible for being able to produce either phrasing, and clearly one will be easier than the other. Indeed, how is the other to be generated at all, since the intension of one of the words is lost? That is, if we use the representation in figure 6.1, we have lost the intension of "afraid," making it difficult to describe the elephants as ones "that are afraid of mice." Somehow, the connection between **afraid** and **fear(x,y)** must be made available to the Generator, perhaps simply as:

$$\textbf{afraid} \equiv \lambda\, x\ \exists\, y\ \textbf{fear}(x, y)$$

The second problem is merely one of consistency: a KR ought to have canonical representations for objects, so that the KR and the programs that use it can be simpler. Therefore,

[1] James Pustejovsky has told me that "fear" and "afraid" may actually be different, for "afraid" seems to create an opaque context (you can be afraid of things that do not exist) which "fear" does not. For instance:

1. I am afraid of ghosts.
2. ? I fear ghosts.

If "fear" and "afraid" are different, my example is wrong, but not necessarily my point. There may still be a problem with the same concept being represented two different ways. Consider a different pair of words, such as the verb "sleep" and the adjective "asleep."

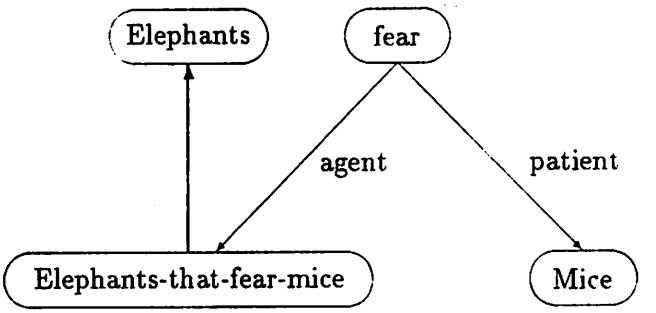

Figure 6.1: A definition of "Elephants that fear mice."

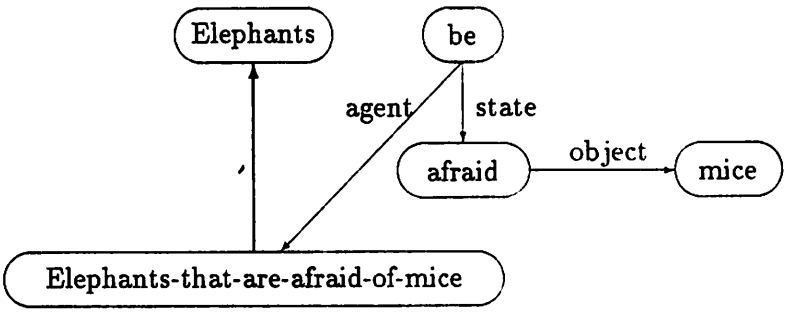

Figure 6.2: A possible definition of "Elephants that are afraid of mice."

whatever representation we pick for **afraid** versus **fear**, we ought to use the same representation for **dead** versus **kill**, **asleep** versus **sleep**, **hidden** versus **hide**, **rotten** versus **rot**, and every other adjective-verb pair.

Chapter 7

Conclusion

The first step towards an AI program that can articulate its thoughts is to have a shared source for those thoughts and their articulations. From that source—the knowledge base—the reasoning component gets concepts and inference rules, and the Generator gets words. This paper has presented some principles for constructing knowledge bases, principles that help the Generator without impairing the reasoning component.

The Generator needed Sorts, as the intensions of common nouns, both simple ones like "elephant" and complex ones like "small pink elephants that fear mice." Distinguishing Sorts from unary predicates turns part of the knowledge base into an AKO ("a kind of") hierarchy, which helps the Generator look for alternate words for concepts, especially ones for which no word may exist. Searching for related words led to viewing the roles of a concept as *dimensions* in a search space, allowing the Generator to try to express different aspects of some concept in a single word. We noted the need for a "Rhombus Rule," so that the Generator can rely on the dimensions of the search space being independent. Also, we used the view of roles as independent dimensions to help answer the question, "how many kinds of rocks are there?" posed by Brachman, Fikes, and Levesque.

The introduction of Sorts motivated a change in the syntax of quantifications to one with two clauses: a restrictive clause that contains a quantifier and a Sort, and a clause that depends on the first. The use of a Sort in the restrictive clause suggested attaching quantificational rules as inheritable data of the Sort. We also saw how quantifications in the two-clause form easily admit extensions to the set of quantifiers, and suggested extending the set of quantifiers to include an odd quantifier, **normal**, which can be used to represent the vast amount of information that is default or prototypical. Reasoning with the extended set of quantifiers—even ones like **most**, let alone **normal**—entails the use of non-monotonic logic, since the inferences are no longer truth-preserving, but merely plausible.

The possibility of writing quantifications in the predicate calculus and attaching them to concepts allowed us to distinguish definitional information from attributive information. The former would be stated in the structure of the network, and the latter attached to it. This added **all** to our set of quantifiers; it had previously seemed superfluous, since the network could easily represent universal quantifications.

With the principle that the network only represents definitional information, we explored definitions of subconcepts as the intensions of noun-phrases using adjectives. In order to avoid making superconcepts of unary intersective predicates, we turned them into value restrictions

on roles. We noted that the definition of a subsective predicate is relative to some Sort, yet another use for that notion. We briefly looked at alienating predicates, but many loose ends remain. Finally, we confronted the fact that we may not always be able to turn a unary predicate into a related binary one. One example is "talented." We considered two solutions to the predicament, neither entirely satisfactory.

Subconcepts can also be defined by using verb frames, which results in noun phrases with restrictive relative clauses. We considered some examples of these, but did not go into the many details, such as tense and aspect, adverbs, subcatagorization frames, and so forth.

Lastly, we considered a problem with this whole approach—that it may cut up the world into small pieces without retaining the connectivity and fluidity of concepts and words. We should not multiply concepts endlessly, but rather look for canonical representations and find ways of expressing them in different ways. Not every English phrase will derive from a different object in a knowledge representation. Still, this problem does not seem insurmountable: the relationships of concepts and words is surely representable.

I have only attacked a small part of the whole problem. I have looked only at noun phrases, and even there I have not considered problems of noun-noun modification, possessives, appositives, and determiners other than quantifiers. A great deal of work still needs to be done. Nevertheless, I believe that the principles and suggestions described in this paper are sound and will result in knowledge representations that are clear, have well-defined semantics, and can eventually be used to make programs that can tell us, in plain English, what is in a knowledge base.

Acknowledgements

I thank David D. McDonald for countless invaluable discussions, for motivating me to really understand what I am talking about, and for endless patience. I thank James Pustejovsky for suggesting I look at Gupta's work on sortal logic and for helping me understand semantics. I appreciate John Brolio, David Lewis, and Dan Suthers for the helpful discussions I had with them. Finally, I thank Beverly Woolf for her support and help.

Chapter 8

Bibliography

[1] Ronald J. Brachman. On the Epistemological Status of Semantic Networks. In Ronald J. Brachman and Hector J. Levesque, editors, *Readings in Knowledge Representation.* Morgan Kaufmann Publishers, Inc., 95 First Street, Lost Altos, CA, 1985. Appeared in *Associative Networks: Representation and Use of Knowledge by Computers* 3–50, edited by N. V. Findler, New York: Academic Press, 1979.

[2] Ronald J. Brachman, Richard E. Fikes, and Hector J. Levesque. Krypton: A Functional Approach to Knowledge Representation. *Computer*, 16(10):67–73, October 1983.

[3] G. W. Cottrell. A Model of Lexical Access of Ambiguous Words. In *Proceedings of AAAI-84.* AAAI, 1984.

[4] Charles Dickens. *Hard Times.* Penguin Books, 1856.

[5] H. P. Grice. Logic and Conversation. In A. P. Martinich, editor, *The Philosophy of Language.* Oxford University Press, 1985.

[6] Anil Gupta. *The Logic of Common Nouns: An Investigation in Quantified Modal Logic.* Yale University Press, New Haven and London, 1980.

[7] Hector J. Levesque and Ronald J. Brachman. A Fundamental Tradeoff in Knowledge Representation and Reasoning (Revised Version). In Ronald J. Brachman and Hector J. Levesque, editors, *Readings in Knowledge Representation.* Morgan Kaufmann Publishers, Inc., 95 First Street, Lost Altos, CA, 1985.

[8] E. Rosch and B. B. Lloyd. *Cognition and Categorization.* Lawrence Erlbaum Associates, Hillsdale, NJ, 1978.

[9] E. Rosch, C. B. Mervis, W. D. Gray, D. M. Johnson, and P. Boyes-Braem. Basic Objects in Natural Catagories. *Cognitive Psychology*, 8:382–439, 1976.

[10] Roger C. Schank and Christopher K. Riesbeck, editors. *Inside Computer Understanding: Five Programs Plus Miniatures.* Lawrence Erlbaum Associates, Hillsdale, NJ, 1981.

[11] Brian C. Smith. Prologue to 'Reflection and Semantics in a Procedural Language'. In Ronald J. Brachman and Hector J. Levesque, editors, *Readings in Knowledge Representation*. Morgan Kaufmann Publishers, Inc., 95 First Street, Lost Altos, CA, 1985. Prologue to Smith's Ph.D. thesis M.I.T. 1982; also Tech Report MIT/LCS/TR-272.

Appendix A

Bitser

Bitser is the small program which takes expressions using a KR of the type I have described and builds input expressions to Mumble-86. Because I have emphasized the representation of nominal terminological concepts, Bitser concentrates on generating definitions.[1]

The following chapters are the actual unedited files of the program.[2] I've tried to write clear Common Lisp code, with comments, so that you can read with a skeptical eye exactly how the program works. A demonstration of Bitser also appears in a later appendix. The most interesting files are undoubtedly `KB`, which is the knowledge base, `DATA`, which is the annotations to the knowledge base, and `Bitser`, which has all the program code. The other files may be interesting only if you want to get a copy of Nikl and Mumble-86 and actually run the program. (I also think the decision-point code is rather nice; perhaps someone else would like to use the utility.)

The Bitser files should be loaded in the following order:[3]

package
nikl-patches
mumble-patches
decision
utils
kb
data
bitser

In addition, I recommend setting the following variables:

---

[1]The name "bitser" derives from the name of the boy in Dickens' *Hard Times* who regurgitated textbook definitions for the schoolmaster Gradgrind: "Horse: Quadruped, Graminivorous, with 42 teeth ..." [4, p. 50]. It is not an acronym.

[2]Bitser runs in Symbolics Genera 7.2. Bitser relies on Nikl (which relies on Clisp) and on Mumble-86. I got the Nikl and Clisp code from Bolt, Beranek and Newman, thanks to Marc Vilain: Clisp version 1.0, Nikl version 4.0. The version of Mumble-86 was courtesy of the Computational Linguistics research group, Computer and Information Science Department, University of Massachusetts at Amherst. The version is dated July 24, 1988. I used the `'standard-windowless-mumble` version. I have not included any code from these systems.

[3]Actually, there is little load-order dependency, with the exception of the package definition, and so other orders will probably work. But this order works, so why fix it?

```
(setq mumble::*window-code?* nil)
(setq tracking::*tracking-system-on?* nil)
```

The first variable must be set because some of Mumble's window-code is incompatible with Symbolics Genera 7.2 (which is why I used the windowless version). The second variable is not strictly necessary, but the Tracker is just for demonstrating Mumble's operation, and therefore is superfluous when running Bitser.

1 package

```
;;; -*- Syntax: Common-lisp; Mode: LISP; Package: USER -*-

;; Without the following, there is a conflict when the bitser package is made,
;; since it inherits 'self from both NIKL and SCL.

(unexport 'nikl::self 'nikl)

(make-package 'bitser
              :use '(nikl symbolics-common-lisp))

(import mumble::
        '(mumble
           ;; argument datatype
           bundle-specification
           specification
           word
           pronoun
           ;; mumble's output stream
           *mumble-text-output*
           ;; object lookup functions
           word-for-string
           pronoun-named
           realization-class-named
           ;; object accessing functions
           name
           parameters
           ;; template/specification functions
           np
           common-np
           proper-np
           link-to-underlying-object
           add-accessory
           make-a-kernel
           make-a-kernel-with-keywords
           clause
           discourse-unit
           make-number-agree
           add-specializing-descriptor
           ;; arguments to template/specification functions
           neuter
           masculine
           feminine
           third
           kind
           plural
           singular
           present
           no-determiner
           ;; Pronouns
```

```
    this
    these
    ;; rclasses
    ako
    predication_to-be
    intransitive-verb_agent-explicit
    transitive-verb_two-explicit-args
    transitive-prepcomp_two-explicit-args
    for-dative_three-explicit-args
    )
'bitser)
```

2 nikl-patches

```
;;; -*- Syntax: Common-lisp; Mode: LISP; Package: NIKL; Patch-File: Yes -*-

#|

This file fixes some problems I had with using the version of NIKL that I had.
It's also a good time to mention some other problems I had, namely:

NKPPConcept does not exist.

According to the manual, the following should work, but it doesn't.  The roles
need to be defined first.

(defconcept human
            (specializes animal)
  (role leg (differentiates limb) (number 2))
  (role arm (differentiates limb) (number 2)))

According to the manual, successive definitions of a concept accumulate
information.  Therefore, the following should work (this is an example from the
manual):

(defconcept physobj
            primitive
            (specializes thing))

(defconcept animal
            primitive
            (specializes physobj))

(defconcept animal (role limb (vrconcept physobj)))

(defconcept animal (role limb (max 4) (min 0)))

Unfortunately, when I do this, it forgets that animal specializes physobj, and
decides that it specializes THING (the default, since THING is the most general
concept).

|#

;;; ============================================================================

;;; This function was taken from the file >bbn>nikl>niklinclude. When used, it
;;; produces an error because of an insufficient number of arguments to the
;;; function CKLONEMarkAsChanged.  This adds a dummy third argument.

(DEFUN CLISP:DEFDISJOINT (OptionalName BranchClassNames)

    (ASSERT (for Names in BranchClassNames always (AND Names
                                                       (symbolp Names)))
```

```
        )
   (PROG ((DisjointConcepts (for Name in BranchClassNames
                              collect (FindOrCreateConceptWithName
                                        Name)))
          (DisjointnessClass (CKLONEGetObjectWithName OptionalName
                                                      'Disjointness))
          Record)
         (cond
           (DisjointnessClass (NKConceptDisjointClass
                                              DisjointnessClass
                                               DisjointConcepts))
           (T (SETQ DisjointnessClass (NKConceptDisjointClass
                                              DisjointnessClass
                                               DisjointConcepts))
              (cond
                (OptionalName (SETQ Record (
                                  CKLONEFOMNameToObjectRecord
                                  OptionalName
                                  'Disjointness))
                              (/replace CKLONEDefinition of Record
                                 with (append (LIST 'DEFDISJOINT
                                                    OptionalName)
                                              (list BranchClassNames)))
                              (CKLONESetObjectName DisjointnessClass
                                                   OptionalName)
                              (CKLONEMarkAsChanged OptionalName
                                                   'Disjointness
                                                   'reason        ; Patch here
                                                   )))))
          (RETURN DisjointnessClass)))

;;; ============================================================================

;;; I'm not certain this is a bug, but it seems to be.  This function, defined
;;; in the nkconcepts file, executed the MAPHASH even when the DerivedClasses
;;; was nil, which signaled an error.  Even if this was an isolated event, my
;;; WHEN clause will merely be unnecessary, not erroneous.

(DEFUN \#NKDeleteDisjointClass (&OPTIONAL Class &REST IGNORE)

  (PROG ((DerivedClasses (fetch KRDisjointDerivedClasses
                            of Class)))
        (/replace KRDisjointIndex of Class with NIL)
        (/replace KRDisjointBranches of Class with NIL)
        (/replace KRDisjointDerivedClasses of Class with NIL)
        (/replace KRDisjointBasedOnClass of Class with NIL)
        (/replace KRDisjointBasedOnRole of Class with NIL)
        (when DerivedClasses                        ; Patch here
          (MapHash (FUNCTION (LAMBDA (Class Role)
                               (\#NKDeleteDisjointClass Class)))
                   DerivedClasses))
```

```
        (SETQ NKDisjointClassList (SafeDRemove Class
                                               NKDisjointClassList))))

;;; ============================================================================

;;; This function, defined in the nkroles file, called CKLONEGetAliasesForObject
;;; with two arguments when that function only takes one.  I have just commented
;;; out that second argument.

(DEFUN \#NKDeleteRole (&OPTIONAL Role FastFlg &REST IGNORE)

  (PROG ((Domain (NKGetRoleDomain Role))
         (Range (NKGetRoleRange Role)))
        ;;
        ;;(for Dis in (NKFindRoleDisjointnessClasses Role) do
        ;;(#NKRemoveRoleFromDisjointClass Role Dis)) This
        ;;needs to be finished.

        (KRMap
          KRRoleParents Role
          (FUNCTION (LAMBDA (Parent)
                      (/replace KRRoleChildren of Parent
                                with
                                (cond
                                  (FastFlg (SafeDRemove Role (fetch KRRoleChildren
                                                                 of Parent)))
                                  (T (NKFindMostGeneralRoles
                                       (Nconc (SafeDRemove Role
                                                           (fetch KRRoleChildren
                                                                  of Parent))
                                              (fetch KRRoleChildren of Role)))))))))
        (KRMap
          KRRoleChildren Role
          (FUNCTION (LAMBDA (Child)
                      (/replace KRRoleParents of Child
                                with
                                (OR
                                  (cond
                                    (FastFlg (SafeDRemove Role
                                                          (fetch KRRoleParents
                                                                 of Child)))
                                    (T (NKFindMostSpecificRoles
                                         (Nconc (SafeDRemove Role
                                                             (fetch KRRoleParents
                                                                    of Child))
                                                (fetch KRRoleParents of Role)))))
                                  (list NKMostGeneralRole))))))
        (NKMapSubRoles Role (FUNCTION (LAMBDA (SubR)
                                        (KRRemove KRRoleAncestors SubR Role)
                                        (KRRemove KRRolePrimitiveClasses SubR Role)
                                        )))
```

```
        ;;
        ;;(KRMap KRRoleRestrictions Role
        ;;(FUNCTION (LAMBDA (Rest) T))) Should remove each
        ;;restriction from its concept and all below that
        ;;inherit it.

        ;;* Need to find all RoleConstraints of this role and
        ;;remove them.

        (cond
          ((type? KRRoleRecord (fetch KRRoleInverse of Role))
           (/replace KRRoleInverse of (fetch KRRoleInverse
                                              of Role)
                     with NIL)))
        (CKLONEUnnameObject Role 'Role)
        (for Name in (CKLONEGetAliasesForObject Role #| 'Role |# )        ;Patch here
             do (UNMARKASCHANGED Name 'NKROLE))
        (/replace KRRoleAncestors of Role with NIL)
        (/replace KRRoleParents of Role with NIL)
        (/replace KRRoleChildren of Role with NIL)
        (/replace KRRoleData of Role with NIL)
        (/replace KRRoleIData of Role with NIL)
        (/replace KRRolePrimitiveClasses of Role with NIL)
        (/replace KRRoleDisjointnessBranches of Role with NIL)
        (/replace KRRoleRestrictions of Role with NIL)
        (/replace KRRoleCovers of Role with NIL)
        (/replace KRRoleDomain of Role with NIL)
        (/replace KRRoleRange of Role with NIL)
        (/replace KRRoleRestrictions of Role with NIL)
        (/replace KRRoleInverse of Role with NIL)
        (KRRemove KRConceptInverseRoleDomains Domain Role)
        (KRRemove KRConceptInverseRoleRanges Range Role)
        (RETURN 'DELETED)))

;;; ============================================================================

;;; This isn't really a bug, just a mis-feature.  This function was taken from
;;; the file >bbn>nikl>nkroles.  Originally, it had code to test that the
;;; inverse-role was actually null, before allowing the replacement.  Since
;;; there is no other function that lets you set the inverse-role, it was
;;; impossible to re-define the inverse-role or even re-evaluate the same one.
;;; I'm hoping that the restriction was not crucial.

(DEFUN NKInverseRole (&OPTIONAL Role InvRole &REST IGNORE)

    (ASSERT (AND (type? KRRoleRecord Role)
                 (type? KRRoleRecord InvRole)
                 ;; Patch:  commented the following out
                 ;;(NULL (fetch KRRoleInverse of Role))
                 ;;(NULL (fetch KRRoleInverse of InvRole))
                 ))
```

```
    (/replace KRRoleInverse of Role with InvRole)
    (/replace KRRoleInverse of InvRole with Role))

;;; ============================================================================

;;; The following are not patches to functions; rather, they do initializations
;;; that the NIKL folks seem to have forgotten to do.  This should be done
;;; before any calls to NIKL functions.

(NKInitializeTaxonomies)

;;; DFNFLG is some random variable from the CLISP stuff.  It didn't get
;;; initialized.  I hope this sets it right.

(CKLONEResetFlags)

;;; This function is the only one which initializes the CKLONESynonymHashTable
;;; (which is used in CKLONEParseConceptForm, called from DefConcept), yet it
;;; isn't called, so I call it.  I hope this is right.

(CKLONEConfigure)
```

3 mumble-patches

```
;;; -*- Syntax: Common-lisp; Mode: LISP; Package: MUMBLE; Patch-File: Yes -*-

;;; ============================================================================
;;; Mode setting.  Isn't a patch; I just didn't want to be hassled by queries.

(setq *ask-user-about-ambiguous-choice-sets?* nil)

;;; ============================================================================
;;; PATCHES

;;; ================ Message Level modifications

;;; Modified to allow realization functions (not just the names) as arguments.
;;; Bitser usually selects the realization function itself, and so it doesn't
;;; need this function to do the lookup.

(defun MAKE-A-KERNEL (realization-function-or-name &rest arguments)
  (let* ((rfon   realization-function-or-name)
         (types  '(curried-realization-class realization-class single-choice))
         (rfun   (etypecase rfon
                   (curried-realization-class  rfon)
                   (realization-class          rfon)
                   (single-choice              rfon)
                   (symbol                     (car (mumble-values rfon types))))))
    (unless rfun
      (mbug "~A is not the name of any defined realization function" rfon))
    (let ((k  (make-kernel-specification :realization-function rfun)))
      (apply #'set-kernel-arguments k arguments))))

;;; I think I modified this just for cosmetic reasons.

(defun SET-KERNEL-ARGUMENTS  (kernel &rest raw-arguments)
  "Toplevel form to be used in Templates"
  (flet ((postprocess-arg (arg)
           (typecase arg
             (string  (word-for-string arg))
             (null    (cerror "trust me; it's right"
                              "NIL is probably an incorrect argument to ~s"
                              kernel))
             (keyword (instantiate-mapping arg))
             (symbol         arg)
             (specification  arg)
             (word           arg)
             (pronoun        arg)
             (otherwise      (instantiate-mapping arg)))))
    (set-arguments kernel (mapcar #'postprocess-arg raw-arguments)))
  kernel)

;;; See the documentation string of this one.  The Bitser function
```

```
;;; EXPRESS-PREDICATE has argument plists in this form, and therefore it uses
;;; this function.

(defun MAKE-A-KERNEL-WITH-KEYWORDS
       (realization-class underlying-object &rest argument-plist)
  "Like MAKE-A-KERNEL, but it doesn't require the arguments to be in the order
that the realization class takes them.  Instead, it iterates over the formals
of the realization class, and gets the matching actual from the argument-plist."
  (check-type realization-class realization-class)
  (flet ((get-arg (formal)
           ;; FORMAL is actually a Mumble 'parameter' object
           (or (getf argument-plist (intern (symbol-name (name formal)) 'keyword))
               (error "no ~s argument to ~s" formal realization-class))))
    (make-kernel-specification
      :underlying-object underlying-object
      :realization-function realization-class
      :arguments (mapcar #'get-arg (parameters realization-class)))))

;;; I think I modified this just for cosmetic reasons.

(defun ADD-ACCESSORY (bundle name &optional value)
  (check-type bundle bundle-specification)
  (etypecase name
    (accessory-type name)
    (keyword        (setq name (accessory-type-named name)))
    (symbol         (mbug "Accessory types are named by keywords~%~
                           you used a symbol for ~A in ~A"
                          name bundle)))
  (etypecase value
    (null            nil)
    (symbol          (setq value (accessory-value-named value)))
    (accessory-value value)
    (specification   value)
    (word            value)
    (string          (setq value (word-for-string value))))
  (set-accessories bundle (acons name value (accessories bundle))))

;;; ================ modifications for pronouns

;;; Used to not allow pronouns as arguments to phrases.

(defun CREATE-PHRASE-PARAMETER-ARGUMENT-LIST (parameter-list argument-list)
  (mapcar #'(lambda (parameter argument)
              (when (listp argument)
                (setq argument (reduce-argument argument)))
              (cons parameter
                    (etypecase argument
                      (word          argument)
                      (pronoun       argument)
                      (trace         argument)
                      (parameter     (return-rclass-parameter-value argument))
```

```
                    (specification argument))))
  parameter-list argument-list))

;;; Used to break if it could not determine the case from the labels, even in the
;;; situation when the case did not affect the morphology.

(defun PRONOUN-MORPHOLOGY (pronoun local-labels)
  (let ((cases     (cases pronoun)))
    (if (stringp cases)
        (send-to-output-stream (the string cases))
        (let* ((desired-case   (determine-case-from-labels local-labels))
               (actual-pronoun
                 (case desired-case
                   (nominative     (pronoun-cases-nominative    cases))
                   (objective      (pronoun-cases-objective     cases))
                   (genitive       (pronoun-cases-genitive      cases))
                   ;; (reflexive      (pronoun-cases-reflexive     cases))
                   ;; that line is deleted
                   (possessive-np  (pronoun-cases-possessive-np cases))
                   (otherwise      (mbug "Unexpected case ~s for pronoun ~s"
                                         desired-case pronoun)))))
          (send-to-output-stream (the string actual-pronoun))))))

;;; ================ modifications for doing DETERMINERS (quantifiers)

;;; Had to modify the following code in order to allow quantifiers as
;;; determiners.  For example, in "All cows eat grass" ``all'' is a
;;; determiner.

(define-slot-label DETERMINER)

(define-splicing-attachment-point DETERMINER
  reference-labels (ugly)
  link (first)
  new-slot (determiner))

(define-node-label NP
   word-stream-actions ((determiner initial))
   associated-attachment-points (possessive determiner))

(define-accessory-type :DETERMINER-POLICY
   (indefinite-first-mention_definite-subsequent-mentions
    always-definite
    no-determiner
    anonymous-individual
    known-individual
    kind
    (determiner)))

(defun PROCESS-DETERMINER-ACCESSORY (bundle np determiner-policy)
  (check-type bundle bundle-specification)
```

```
  (check-type np      node)
  (check-type determiner-policy (or accessory-value word))
  (etypecase determiner-policy
    (word             (process-word-determiner bundle np determiner-policy))
    (accessory-value  (set-determiner-state
                       np
                       (ecase (name determiner-policy)
                         (indefinite-first-mention_definite-subsequent-mentions
                           (if (first-mention? (underlying-object bundle))
                               'indefinite
                               'definite))
                         (always-definite              'definite)
                         (no-determiner            'no-determiner)
                         (anonymous-individual           'indefinite)
                         (known-individual             'definite)
                         (kind                     'indefinite))))))

;;; We analyze determiners as being spliced into the NP phrase.  We also
;;; set the NP determiner-state to be no-determiner, so that we don't
;;; get "a" or "the," too.

(defun PROCESS-WORD-DETERMINER (bundle np determiner)
  (check-type bundle     bundle-specification)
  (check-type np         node)
  (check-type determiner (or symbol word))
  (set-determiner-state np 'no-determiner)
  (let* ((ap           (splicing-attachment-point-named 'determiner))
         (context      (context-object np))
         (position     (cdr (assoc ap (available-aps context))))
         (new-position (attach-by-splicing ap position determiner)))
    (delete! ap (available-aps context))
    (push (cons 'determiner new-position)
          (position-table context))))

;;; ================ modifications for doing 'a kind of' constructions

(define-splicing-attachment-point REQUIRED-ARG
  reference-labels (ugly)
  link (next)
  new-slot (of-complement))

(define-slot-label NP-HEAD
  associated-attachment-points
  (np-prep-complement
    non-restrictive-relative-clause
    non-restrictive-appositive
    required-arg))

(define-phrase N-BAR (predicate arg)
  (np
    np-head       predicate
```

```
    of-complement arg))

(define-realization-class AKO (superclass classification-word)
  ;; can't do this as a single-choice because it has two args
  ((n-bar classification-word superclass)
   :grammatical-characteristics (np)))

;;; ============================================================================
;;; Additional Mumble functions (that is, these are not patches)

(defun MAKE-NUMBER-AGREE (np1 np2)
  (check-type np1 specification)
  (check-type np2 specification)
  (let* ((number-acc  (accessory-type-named :number))
         (np1-number  (cdr (assoc number-acc (accessories np1)))))
    (if (null np1-number)
        (error "~s has no number, so can't make agreement." np1)
        (add-accessory np2 number-acc np1-number))))

(defun HEADED-BY (bundle head)
  (let ((postprocessed-head  (etypecase head
                               (specification  head)
                               (word           head)
                               (string         (word-for-string head)))))
  (set-bundle-head bundle postprocessed-head)
  bundle))

(defun NP (&optional head)
  (let ((b  (make-a-bundle 'general-np)))
    (when head
      (headed-by b head))
    b))

(defun CLAUSE (&optional head)
  (let ((b  (make-a-bundle 'general-clause)))
    (when head
      (headed-by b head))
    b))

(defun DISCOURSE-UNIT (&optional head)
  (let ((b  (make-a-bundle 'discourse-unit)))
    (when head
      (headed-by b head))
    b))

(defun COMMON-NP (&rest args)
  "Takes the arguments to an np-common-noun kernel and returns an NP bundle"
  (np (apply #'make-a-kernel 'np-common-noun args)))

(defun PROPER-NP (&rest args)
  "Takes the arguments to an np-proper-name kernel and returns an NP bundle"
```

```
  (np (apply #'make-a-kernel 'np-proper-name args)))

;;; ============================================================================
;;; Words

(define-word "thing"       (noun))
(define-word "kind"        (noun))
(define-word "order"       (noun))
(define-word "class"       (noun))
(define-word "genus"       (noun))
(define-word "species"     (noun) plural "species")
(define-word "animal"      (noun))
(define-word "mammal"      (noun))
(define-word "rodent"      (noun))
(define-word "rat"         (noun))
(define-word "human"       (noun))
(define-word "man"         (noun) plural "men")
(define-word "elephant"    (noun))
(define-word "pachyderm"   (noun))

(define-word "property"    (noun))
(define-word "activity"    (noun))
(define-word "runner"      (noun))
(define-word "run"         (verb))
(define-word "bite"        (verb))
(define-word "employ"      (verb))
(define-word "kiss"        (verb))
(define-word "slap"        (verb))
(define-word "write"       (verb))

(define-word "gray"        (adjective))
(define-word "bank"        (noun))

(define-word "sex"         (noun))
(define-word "female"      (adjective))
(define-word "male"        (adjective))

(define-word "occupation"  (noun))
(define-word "comedy"      (noun))

(define-word "hobby"       (noun))
(define-word "philately"   (noun))

(define-word "person"      (noun) plural "people")
(define-word "woman"       (noun) plural "women")
(define-word "comic"       (noun))
(define-word "comedienne"  (noun))

(define-word "philatelist" (noun))

(define-word "blonde"      (noun))
```

```
(define-word "brunette"     (noun))
(define-word "redhead"      (noun))

(define-word "rock"         (noun))

(define-word "grass"        (noun))
(define-word "tusk"         (noun))

(define-word "eat"          (verb))

(define-word "like"         (verb))
(define-word "dislike"      (verb))
(define-word "detest"       (verb))
(define-word "admire"       (verb))

(define-word "no"           (determiner))
(define-word "one"          (determiner))
(define-word "two"          (determiner))
(define-word "three"        (determiner))
(define-word "four"         (determiner))
(define-word "five"         (determiner))
(define-word "six"          (determiner))
(define-word "seven"        (determiner))
(define-word "eight"        (determiner))
(define-word "nine"         (determiner))
(define-word "ten"          (determiner))

(define-word "every"        (determiner))
(define-word "each"         (determiner))
(define-word "all"          (determiner))
(define-word "normal"       (determiner))
(define-word "most"         (determiner))
(define-word "some"         (determiner))
(define-word "several"      (determiner))
(define-word "few"          (determiner))

(define-word "Clyde"        (proper-noun))
(define-word "Stephanie"    (proper-noun))
(define-word "Chris"        (proper-noun))

(define-pronoun these
  person third
  gender neuter
  number plural
  cases "these")
```

4 decision

```
;;; -*- Syntax: Common-lisp; Mode: LISP; Package: BITSER -*-

#|

When we write a prototype program, we often want to leave certain parts uncoded.
For instance, it may have to make a decision between a definite or an indefinite
article, and we don't know how to do that, so we don't want to write any code.
We would rather put it a notation that this is a decision point, and have the
program ask the user if it doesn't know how to make the decision.  The following
code lets us annotate programs with decision points.  The program can then be
run in three different modes:

1) ring the changes.  That is, run the program enough times so that every
question is answered both yes and no, so that we can see all the possibilities.
It does not try to decide how the question should be answered; it just answers
both ways.

2) Ask questions.  Don't ever let the program try to figure things out for
itself, query the user instead.

3) Figure it out.  If it can, the program should answer questions for itself.
It will only ask the user if a decision procedure is not known.

|#

;;; ============================================================================

(deftype decision-mode () '(member :ring-changes :ask-questions :figure-it-out))

(defvar *decision-mode* :figure-it-out
  "Tells decision points how to execute.")

(proclaim '(type decision-mode *decision-mode*))

;;; ============================================================================

(defvar *decision-results*
        (make-array 100 :adjustable t :fill-pointer 0)
  "When we're ringing the changes, we need to know what decisions we made last
time so that we can make a different one this time.  Essentially, this records
the decisions made on a path from the root to a leaf of the decision tree.")

(defvar *decision-number*
        -1
  "This variable tells us where we are as we re-trace the decision path used
last time.  We will only want to vary the last left branch step.")

(defvar *decision-to-vary*
        -1
```

```
  "This variable is computed as the end of each run when ringing the changes.
It indicates which decision we will vary then next time through, and it's the
last 'left' or NIL branch.  (left = NIL, right = T, and we always go left when
exploring a new subtree of the decision tree.")

(defun possibly-grow-decision-array ()
  (let ((len (length *decision-results*)))
    (unless (< *decision-number* len)
      (setq *decision-results* (adjust-array *decision-results* (+ len 100))))))

(defun make-decision ()
  (possibly-grow-decision-array)
  (cond ((< *decision-number* *decision-to-vary*)
         (aref *decision-results* *decision-number*))
        ((= *decision-number* *decision-to-vary*)
         T)
        ((> *decision-number* *decision-to-vary*)
         NIL)))

(defun record-decision (decision)
  (setf (aref *decision-results* *decision-number*) decision)
  (incf *decision-number*))

;;; ============================================================================

(defun decision-point (name &rest args)
  (let ((q  (get name 'decision-question)))
    (unless q (error "~s is not the name of a decision point." name))
    (fresh-line *standard-output*)
    (ecase *decision-mode*
      (:figure-it-out (if (get name 'decision-function?)
                          (let ((d  (apply name args)))
                            (apply #'format t q args)
                            (format t " =====> ~s~%" d)
                            d)
                          (apply #'y-or-n-p q args)))
      (:ask-questions (apply #'y-or-n-p q args))
      (:ring-changes  (let ((d  (make-decision)))
                        (record-decision d)
                        (apply #'format t q args)
                        (format t " =>  ~s~%" d)
                        d)))))

;;; ============================================================================

(defmacro define-decision-point (name question &optional arglist &body function)
  (check-type name     symbol)
  (check-type question string)
  (if (not (char-equal (char question (- (length question) 1)) #\space))
      (cerror "go ahead anyway"
              "The string ~s does not end in a space.~%~
```

```
              Questions look nicer when they do."
             question))
  (if (null function)
      `(progn (setf (get ',name 'decision-question) ,question)
              (setf (get ',name 'decision-function?) nil)
              ',name)
      `(progn (setf (get ',name 'decision-question) ,question)
              (setf (get ',name 'decision-function?) t)
              (defun ,name ,arglist ,@function))))

;;; ============================================================================

(defun figure-out-decision-to-vary ()
  "We always go back to the last NIL and we will change it to a T next time."
  (let ((result nil))
    (do ((dn (1- *decision-number*) (1- dn)))
        ((cond ((< dn 0)
                (setq result :no-more-left-to-vary))
               ((null (aref *decision-results* dn))
                (setq result dn))
               (T nil))
         result))))

(defmacro ring-changes (&body code)
  `(let ((*decision-mode*  :ring-changes))
     (declare (special *decision-mode*))
     (let ((results  nil))
       (do ((*decision-number*  0  0)
            (*decision-to-vary* -1 (figure-out-decision-to-vary)))
           ((eq *decision-to-vary* :no-more-left-to-vary))
         (declare (special *decision-number* *decision-to-vary*))
         (push (print (progn ,@code)) results)
         (format t "~2&"))
       (nreverse results))))

(defmacro ask-questions (&body code)
  `(let ((*decision-mode*  :ask-questions))
     (declare (special *decision-mode*))
     ,@code))

(defmacro figure-it-out (&body code)
  `(let ((*decision-mode*  :figure-it-out))
     (declare (special *decision-mode*))
     ,@code))
```

5 utils

```
;;; -*- Syntax: Common-lisp; Mode: LISP; Package: BITSER -*-

;;; ============================================================================
;;; THEN & ELSE

(defmacro then (&rest forms) `(progn ,@forms))
(defmacro else (&rest forms) `(progn ,@forms))

;;; ============================================================================
;;; Because I can never remember which is destructive.  The exclamation mark
;;; means it's destructive.

(deff remove!        'delete)
(deff remove-if!     'delete-if)
(deff remove-if-not! 'delete-if-not)

;;; ============================================================================
;;; The NIKL functions simply return NIL if they cannot find a concept or role
;;; of the given name.  That's fine, but often I want to signal an error if the
;;; object is not found, so I simply use these.

(defsubst NKGetNamedConcept-error (name)
  "Returns the named NIKL Concept, or signals an error if it's not found."
  (or (NKGetNamedConcept name)
      (error "No such NIKL concept:  ~s" name)))

(defsubst NKGetNamedRole-error (name)
  "Returns the named NIKL Role, or signals an error if it's not found."
  (or (NKGetNamedRole name)
      (error "No such NIKL role:  ~s" name)))

;;; ============================================================================
;;; For function argument lists, the property list format is easy, but for
;;; internal representation, I prefer association lists.  This function simply
;;; converts one to the other, allowing the keys and values to be modified, if
;;; desired.

(defun plist->alist
       (plist &optional (key-function #'identity) (value-function #'identity))
  "Converts a PLIST--a list in property-list format:  (key1 value1 key2 value2 ...)
to an ALIST--a list in association-list format:  ((key1 . value1) (key2 . value2) ...)
In addition, it calls KEY-FUNCTION on each key and VALUE-FUNCTION on each value.
Each of those functions should take exactly one argument.  PLIST->ALIST preserves
the order of the list, and is non-destructive."
  (let ((alist  nil))
    (do* ()
         ((null plist)
          (nreverse alist))
      (setq alist
```

```
            (acons (funcall key-function (pop plist))
                   (funcall value-function (pop plist))
                   alist)))))

;;; ============================================================================
;;; For showing off Bitser without having to use the Mumble window system.

(defun mumble-into-string (message)
  (check-type message specification)
  (with-output-to-string (stream)
    (let ((*mumble-text-output*  stream))
      (declare (special *mumble-text-output*))
      (mumble (list message)))))

;;; ================ Types

(deftype nikl-concept () '(satisfies NKConcept?))

(deftype nikl-role    () '(satisfies NKRole?))

(deftype sort () '(satisfies sort?))

;;; ============================================================================
;;; Types of concepts

(defun verb-concept? (concept)
  (check-type concept nikl-concept)
  (NKSuperC? concept (NKGetNamedConcept-error 'verb)))

(defun sort? (concept)
  (check-type concept nikl-concept)
  (NKSuperC? concept (NKGetNamedConcept-error 'sort)))

(defun substance? (concept)
  (check-type concept nikl-concept)
  (NKSuperC? concept (NKGetNamedConcept-error 'substance)))

;;; ================ Subsumption Predicate

(defun parent-concept? (parent child)
  (check-type parent nikl-concept)
  (check-type child  nikl-concept)
  (member parent (NKImmediateSuperCs child)))
```

6 kb

```
;;; -*- Package: BITSER; Syntax: Common-lisp; Mode: LISP -*-

;;; ============================================================================
;;; Very general concepts

(defconcept  SORT        primitive)
(defconcept  PROPERTY    primitive)
(defconcept  SUBSTANCE   primitive)
(defconcept  VERB        primitive)

;;; ================ Next level down

(defconcept  PHYSOBJ   primitive (specializes sort))

(defconcept  ANIMALS   primitive (specializes physobj)
  (role limb (vrconcept physobj) (max 4) (min 0)))

(defconcept  MAMMALS   primitive (specializes animals))

(defconcept  ELEPHANTS primitive (specializes mammals)
  (role relative-age))

(defconcept  GRASS   primitive  (specializes substance))
(defconcept  TUSKS   primitive  (specializes physobj))

(defconcept  RED     primitive  (specializes property))
(defconcept  BROWN   primitive  (specializes property))
(defconcept  BLONDE  primitive  (specializes property))
(defconcept  GRAY    primitive  (specializes property))

;;; ================ some kinds of elephant

(defconcept  CIRCUS-ELEPHANTS  primitive (specializes elephants))

(defconcept  OLD-ELEPHANTS     primitive (specializes elephants)
  (role relative-age (vrconcept old)))

;;; ================ Polysemy

(defconcept  RIVERBANK       primitive)      ; => bank
(defconcept  FIDUCIARY-BANK  primitive)      ; => bank

;;; ================ Male and Female

(defconcept  SEXES     primitive)
(defconcept  FEMALE    primitive  (specializes sexes))
(defconcept  MALE      primitive  (specializes sexes))

;;; DefDisjoint requires its arguments NOT be taxonomized.
```

```
(DefDisjoint SEX-DICHOTOMY (female male))

;;; ================ Occupations

(defconcept  OCCUPATIONS  primitive)
(defconcept  COMEDY       primitive  (specializes occupations))

;;; ================ Hobbies

(defconcept  HOBBIES      primitive)
(defconcept  PHILATELY          primitive (specializes hobbies))

;;; ================ comedienne

(defconcept  PERSON       primitive  (specializes mammals)
  (role sex        (vrconcept sexes) (number 1))
  (role occupation (vrconcept occupations))
  (role hobby      (vrconcept hobbies))
  (role hair-color)
  (role eye-color))

(defconcept  WOMAN  (specializes person)
  (role sex (vrconcept female)))

(defconcept  COMIC  (specializes person)
  (role occupation (vrconcept comedy)))

;;; The Nikl classifier should make this a subconcept of both WOMAN and COMIC,
;;; even though they're not mentioned.

(defconcept  COMEDIENNE  (specializes person)
  (role sex (vrconcept female))
  (role occupation (vrconcept comedy)))

;;; ================ philatelist

(defconcept  PHILATELISTS  (specializes person)
  (role hobby (vrconcept philately)))

;;; ================ hair color

(defconcept  BLONDES    (specializes woman)
  (role hair-color (vrconcept blonde)))

(defconcept  BRUNETTES  (specializes woman)
  (role hair-color (vrconcept brown)))

(defconcept  REDHEADS   (specializes woman)
  (role hair-color (vrconcept red)))
```

```
;;; ================ rocks

(defconcept  ROCKS  primitive  (specializes phyobj)
  (role mode-of-creation)
  (role relative-size)
  (role color))

(defconcept  IGNEOUS-ROCKS     (specializes rocks)
  (role mode-of-creation (vrconcept volcanic)))

(defconcept  SEDIMENTARY-ROCKS (specializes rocks)
  (role mode-of-creation (vrconcept sedimentation)))

(defconcept  METAMORPHIC-ROCKS (specializes rocks)
  (role mode-of-creation (vrconcept heat-and-pressure)))

(defconcept  LARGE-ROCKS  (specializes rocks)
  (role relative-size (vrconcept large)))

(defconcept  GRAY-ROCKS   (specializes rocks)
  (role color (vrconcept gray)))

;;; 'pet rocks' is tough.  May have to be primitive.

;;; ================ defining thematic roles and their inverses

(defrole theta primitive)

(NKInverseRole (defrole agent    primitive (differentiates theta))
               (defrole agent-   primitive (differentiates theta)))

(NKInverseRole (defrole theme    primitive (differentiates theta))
               (defrole theme-   primitive (differentiates theta)))

(NKInverseRole (defrole patient  primitive (differentiates theta))
               (defrole patient- primitive (differentiates theta)))

(NKInverseRole (defrole goal     primitive (differentiates theta))
               (defrole goal-    primitive (differentiates theta)))

(NKInverseRole (defrole source   primitive (differentiates theta))
               (defrole source-  primitive (differentiates theta)))

;; the following pair of roles is for high arity verbs like BET and COMPARE

(NKInverseRole (defrole theme1   primitive (differentiates theta))
               (defrole theme1-  primitive (differentiates theta)))

(NKInverseRole (defrole theme2   primitive (differentiates theta))
               (defrole theme2-  primitive (differentiates theta)))
```

```
;;; ================ defining verb frames

;;; These probably ought to be AGENT and THEME, but Mumble is already
;;; encoded in terms of PATIENT, so I'm not going to fight it.

(defconcept  INGEST  primitive (specializes verb)
  (role agent)
  (role patient))

(defconcept  EMOTIONAL-ATTITUDE  primitive (specializes verb)
  (role agent)
  (role patient))

(defconcept  HAVE  primitive  (specializes verb)
  (role agent)
  (role patient))

(defconcept  EAT       primitive  (specializes ingest))

(defconcept  LIKE      primitive  (specializes emotional-attitude))
(defconcept  DISLIKE   primitive  (specializes emotional-attitude))
(defconcept  DETEST    primitive  (specializes emotional-attitude))
(defconcept  ADMIRE    primitive  (specializes emotional-attitude))

(defconcept  BET   primitive  (specializes verb)
  (role agent (vr bettors))
  (role patient (vr person))
  (role theme1)
  (role theme2))

;;; ================ Elephants that like people

;;; this is a pain to have to define.  Ought to be another way.

(defconcept |LIKE(ELEPHANTS,PEOPLE)|  (specializes like)
  (role agent   (vrconcept elephants-that-like-people))
  (role patient (vrconcept person)))

(defconcept  ELEPHANTS-THAT-LIKE-PEOPLE  (specializes elephants)
  (role agent- (vrconcept |LIKE(ELEPHANTS,PEOPLE)|)))


;;; ============================================================================

(NKClassifyAll)

;;; NKConceptCovering requires its argument to be a list of TAXONOMIZED
;;; concepts.

(NKCoverConcept (NKGetNamedConcept-error 'sexes)
                (NKConceptCovering nil (mapcar #'NKGetNamedConcept-error
```

```
'(female male))))
```

7 data

```
;;; -*- Package: BITSER; Syntax: Common-lisp; Mode: LISP -*-

;;;; This file contains code to annotate (add local data, either IDATA
;;;; or DATA) the NIKL hierarchy.  At the end of the file, it uses that
;;;; code to do some annotation; that is, adding some sample data to our
;;;; sample knowledge base.

;;; ============================================================================
;;; Concept Words
;;; ============================================================================

(defun set-concept-word (concept-name string)
  "Given the name of a concept and a string, it looks up the NIKL concept and
looks up the Mumble word and places the word on the concept as local data."
  (check-type concept-name symbol)
  (check-type string       (or string null function))
  (let ((c  (NKGetNamedConcept-error concept-name)))
    (NKAddConceptData c 'word (if (stringp string)
                                  (word-for-string string)
                                  string))))

(defun word-mapping (concept &optional error?)
  "Returns a word for the concept, and if there is none, optionally signals an
error.  If the 'word' is a function, it calls the function.  Presumably, the
function looks at the environment and decides what word to return."
  (check-type concept nikl-concept)
  (let ((w  (NKGetConceptData concept 'word)))
    (etypecase w
      (word      w)
      (function  (the word (funcall w)))
      (null      (if error? (error "There is no word for ~s" concept))))))

;;; ============================================================================
;;; Putting words on concepts

(set-concept-word 'animals          "animal")
(set-concept-word 'mammals          "mammal")

(defvar *ponderous-pedantic* nil
  "This variable is meant to be emblematic of various rhetorical goals that a
program (such as Ed Hovy's Pauline) could have and which would influence word
choice.")

(set-concept-word 'elephants        #'(lambda ()
                                        (if *ponderous-pedantic*
                                            (word-for-string "pachyderm")
                                            (word-for-string "elephant"))))

(set-concept-word 'grass            "grass")
```

```
(set-concept-word 'tusks            "tusk")

(set-concept-word 'red              "red")
(set-concept-word 'gray             "gray")

;;; demonstrates that polysemy is trivial

(set-concept-word 'riverbank        "bank")
(set-concept-word 'fiduciary-bank   "bank")

(set-concept-word 'sexes            "sex")
(set-concept-word 'female           "female")
(set-concept-word 'male             "male")

(set-concept-word 'occupations      "occupation")
(set-concept-word 'comedy           "comedy")

(set-concept-word 'hobbies          "hobby")
(set-concept-word 'philately        "philately")

(set-concept-word 'person           "person")
(set-concept-word 'woman            "woman")
(set-concept-word 'comic            "comic")
(set-concept-word 'comedienne       "comedienne")

(set-concept-word 'philatelists     "philatelist")

(set-concept-word 'blondes          "blonde")
(set-concept-word 'brunettes        "brunette")
(set-concept-word 'redheads         "redhead")

(set-concept-word 'rocks            "rock")

(set-concept-word 'have             "have")

(set-concept-word 'eat              "eat")

(set-concept-word 'like             "like")
(set-concept-word 'dislike          "dislike")
(set-concept-word 'detest           "detest")
(set-concept-word 'admire           "admire")


;;; ============================================================================
;;; Specialization Words
;;; ============================================================================

(defun set-concept-specialization-word (concept-name string)
  "Given the name of a concept and a string, it looks up the NIKL concept and
looks up the Mumble word and places the word on the concept as local data.  The
word in this case is to describe the specialization relation that this concept
```

```
hold with its parent.  For instance, 'poodles are a BREED of dog,' so the
specialization word for 'poodles' might be 'breed.'  [This could be extended,
via association lists, to deal with multiple parent concepts, but this is
sufficient to demonstrate the representational requirement.]"
  (check-type concept-name symbol)
  (check-type string  (or string null))
  (let ((c  (NKGetNamedConcept-error concept-name)))
    (NKAddConceptData c 'specialization-word (word-for-string string))))

(defun specialization-word (concept)
  "Returns a word denoting the relationship of the concept to its superior(s).
In most cases, it will be 'kind,' since a concept is a kind of a superc.
Other possibilities are 'breed,' as in 'a collie is a breed of dog.'"
  (check-type concept nikl-concept)
  (let ((word   (NKGetConceptData concept 'specialization-word)))
    (if (and word (decision-point 'use-specialization-word? word))
        word
        (word-for-string "kind"))))

(set-concept-specialization-word 'elephants "species")


;;; ============================================================================
;;; Realization Classes
;;; ============================================================================

;;; Realization Classes are added as INHERITABLE data because we assume
;;; that if a verb-concept semantically specializes another, then it
;;; will relate the same number of objects in the same ways, so that the
;;; same realization-class can be used.  This is an open empirical
;;; question.

;;; Note added later:  Realization classes should probably NOT be inheritable.
;;; Consider 'donate' which is a semantic specialization of 'give' and therefore
;;; will inherit its realization class, except that it ought not to:  ''Give the
;;; museum a painting.'' vs. *''Donate the museum a painting.''

(defun set-concept-realization-class (concept-name rclass-name)
  "Given the name of a concept and the name of Mumble realization-class, it
looks up the NIKL concept and the Mumble object and places the realization class
on the concept as local inheritable data.  Presumably, this is a verb-concept
[such as 'like'] and it will have roles that are the thematic roles [and hence
arguments] of the realization class."
  (check-type concept-name symbol)
  (check-type rclass-name  symbol)
  (let ((c  (NKGetNamedConcept-error concept-name))
        (r  (realization-class-named rclass-name)))
    (NKAddConceptIData c 'realization-class r)))

(defun rclass-mapping (concept &optional error?)
  "Returns a Mumble-86 realization class for that concept.  Not all concepts
```

```
will have them, but all verb concepts will have or inherit one.  If there is no
realization class, an error can be signalled."
  (check-type concept nikl-concept)
  ;; since NKFindConceptIData always returns a list, we return the car
  (or (car (NKFindConceptIData concept 'realization-class))
      (if error? (error "~s has no realization class." concept))))

(set-concept-realization-class 'ingest             'TRANSITIVE-VERB_TWO-EXPLICIT-ARGS)
(set-concept-realization-class 'emotional-attitude 'TRANSITIVE-VERB_TWO-EXPLICIT-ARGS)
(set-concept-realization-class 'have               'TRANSITIVE-VERB_TWO-EXPLICIT-ARGS)

;;; ============================================================================
;;; Foldability
;;; ============================================================================

;;; Annotating Roles as to whether they can be folded into the meaning of the
;;; head noun.

(defun set-foldability (role-name yes-or-no)
  "Given the name of a NIKL Role and either of the symbols YES or NO, it adds
that symbol as local data to the role.  This information is used as advice to
the mechanism that searches for words that incorporate [fold in] that role in
their meaning.  For instance, sex is often foldable ('person' becomes 'man' or
'woman'; 'horse' becomes 'stallion' or 'mare'), while eye color is not."
  (check-type role-name symbol)
  (check-type yes-or-no (member yes no))
  (let ((r  (NKGetNamedRole-error role-name)))
    (NKAddRoleData r 'foldable? yes-or-no)))

(defun foldable? (role)
  "Returns T or Nil, so that it can be used in predicates, even though the
information is stored as Yes or No.  It's stored that way so that we can
distinguish not-foldable from not-stored.  If the role is not marked as
foldable, the current default is that the role IS foldable:  we are being
conservative, since the worst that can happen is to uselessly increase the
search space."
  (check-type role nikl-role)
  (case (NKGetRoleData role 'foldable?)
    ((yes)  t)
    ((no)   nil)
    (T      t)))

(set-foldability 'sex 'yes)
(set-foldability 'occupation 'yes)
(set-foldability 'hair-color 'yes)
(set-foldability 'eye-color 'yes)

;;; ============================================================================
;;; Rules
;;; ============================================================================
```

```
;;; Rules are quantifications in our Sortal Logic:  expressions of the form:
;;;                                    [ Q K x ] $\alpha$
;;; In these formulae, Q is a quantifier, K is a Sort, and x is a variable
;;; bound by Q and appearing in $\alpha$.  This is just like any restricted
;;; quantification.

;;; We add rules of this kind to the Concepts (Sorts) in our NIKL hierarchy
;;; that they are about.

(deftype quantifiers ()
  '(member every each all normal most many some several few no))

(deftype quantifier () '(or quantifiers integer))

(defstruct rule
  (quantifier nil :type (or null quantifier))
  (sort       nil :type (or null sort))
  (variable   nil :type symbol)
  expression)

(defun sexp->rule (sexp)
  "Takes a symbolic expression of the following form,

           (:RULE ( <quantifier> <sort> <variable> ) <expression> )

and turns it into a rule structure, including looking up the concept specified
by the name of the sort."
  (let ((ignore       (the (member :rule) (car sexp)))
        (restriction  (the list (cadr sexp)))
        (expression   (caddr sexp)))
    (let ((quantifier  (the quantifier (car   restriction)))
          (sort        (the symbol     (cadr  restriction)))
          (variable    (the symbol     (caddr restriction))))
      (make-rule :quantifier quantifier
                 :sort       (NKGetNamedConcept-error sort)
                 :variable   variable
                 :expression expression))))

(defun add-rule (r)
   "Adds a RULE to the knowledge base, adding it as inheritable data to the Sort
[a NIKL Concept] that is the domain of the quantification."
   (let ((rule   (etypecase r
                   (rule  r)
                   (list  (sexp->rule r)))))
     (let ((concept (rule-sort rule)))
       (NKAddConceptIData concept 'rules rule))))

(defun get-concept-rules (concept)
  (check-type concept nikl-concept)
  (NKFindConceptIData concept 'rules))
```

```
;;; ============================================================================
;;; Some silly rules about Elephants for our sample knowledge base.

(add-rule '(:rule (every  elephants e) (eat     :agent e :patient grass)))

(add-rule '(:rule (normal elephants e) (have    :agent e :patient tusks)))

(add-rule '(:rule (most   elephants e) (like    :agent e :patient person)))

(add-rule '(:rule (some   elephants e) (dislike :agent e :patient person)))

(add-rule '(:rule (few    elephants e) (detest  :agent e :patient person)))

(add-rule '(:rule (no     elephants e) (admire  :agent e :patient person)))
```

8 bitser

```
;;; -*- Syntax: Common-lisp; Mode: LISP; Package: BITSER -*-


(define-decision-point express-catagory-word? "Express catagory word? ")

(define-decision-point use-specialization-word? "Use specialization word ~s? ")

(define-decision-point plural? "Should ~s be plural? ")

(define-decision-point use-singular? "Use singular for ~s? ")

(define-decision-point
  search-for-more-specific-word-anyway?
  "~s already has the word ~s.  Search for a better one anyway? ")

;;; ============================================================================
;;; Word-Mapping, extended to do some inheritance.

(defun word-mapping-for-random-superc (concept)
  "Searches through the super-concepts of a concept in order to find a (general)
word for the concept.  Does not look at the concept itself; you must do that
yourself."
  (check-type concept nikl-concept)
  (labels ((find-word (c)
             (or (word-mapping c)
                 (some #'find-word (NKImmediateSuperCs c)))))
    (some #'find-word (NKImmediateSuperCs concept))))

;;; ============================================================================
;;; Generating Mumble-86 messages from formulas

(defvar *messages-for-variables* nil
  "A mapping between quantificational variables in a formula and the messages
built for them.  As we recursively build the message for the entire formula,
from outside-to-inside, the variable-messages are substituted for each
occurence of the variable.  The mapping is implemented as an a-list, since that
data structure easily implements the right scoping rules.")

(defun get-message-for-variable (var)
  (and (symbolp var)
       (cdr (assoc var *messages-for-variables*))))

(defmacro bind-message-for-variable (var message &body forms)
  `(let ((*messages-for-variables*  (acons (the symbol ,var)
                                           (the bundle-specification ,message)
                                           *messages-for-variables*)))
     (declare (special *messages-for-variables*))
     ,@forms))
```

```
(defun express-formula (sexp)
  "In spirit, this function builds a Mumble-86 message for any formula roughly
written in the predicate calculus.  [I say ``roughly'' because we make minor
syntactic modifications, such as eliminating commas and moving the predicate
inside the parentheses, lisp-style.]  In reality, this function handles a
ludicrously small subset of the intended functionality--just enough to
demonstrate the idea of expressing rules."
  (etypecase sexp
    (keyword sexp)
    (symbol  (or (get-message-for-variable sexp)
                 (express-concept
                   (or (NKGetNamedConcept sexp)
                       (error "~s is neither a variable nor a Concept" sexp)))))
    (rule    (express-rule sexp))
    (cons    (express-predicate sexp))))

(defun express-concept (concept &key (do-number? t))
  "Builds a Mumble message for a concept.  Currently, it only does Sorts, which
result in Common Noun Phrases, but the provision for extending its functionality
is only possible because we can distinguish Sorts from other concepts."
  (check-type concept nikl-concept)
  (cond ((sort? concept)
         (let ((c  (common-np (word-mapping concept t))))
           (link-to-underlying-object c concept)
           (add-accessory c :gender 'neuter)
           (add-accessory c :person 'third)
           (add-accessory c :determiner-policy 'kind)
           (if do-number?
               (add-accessory c :number (if (decision-point 'plural? concept)
                                            'plural
                                            'singular)))
           c))
        ((substance? concept)
         (let ((c  (common-np (word-mapping concept t))))
           (link-to-underlying-object c concept)
           (add-accessory c :gender 'neuter)
           (add-accessory c :person 'third)
           (add-accessory c :number 'singular)
           (add-accessory c :determiner-policy 'no-determiner)
           c))
        ((verb-concept? concept)
         (error "To express ~s, we'd have to nominalize it~%~
                 or allow sentential arguments.  NYI"
                concept))
        (t
         (error "To express ~s, we'd have to nominalize it.  NYI" concept))))

(defun express-rule (rule)
  "Builds a Mumble-86 message to express a quantificational rule."
  (check-type rule rule)
  (bind-message-for-variable (rule-variable rule)
```

```
                          (express-quantified-sort rule
                                                   (rule-quantifier rule)
                                                   (rule-sort rule))
    (express-formula (rule-expression rule))))

(defun express-quantified-sort (rule quantifier sort)
  (check-type quantifier quantifier)
  (check-type sort nikl-concept)
  (let ((c (common-np (word-mapping sort t))))
    (link-to-underlying-object c rule)
    (add-accessory c :gender 'neuter)
    (add-accessory c :person 'third)
    (add-accessory c :determiner-policy (word-for-quantifier quantifier))
    (add-accessory c :number (etypecase quantifier
                               (integer
                                 (case quantifier
                                   (0   (if (decision-point 'plural? sort)
                                            'plural
                                            'singular))
                                   (1    'singular)
                                   (t    'plural)))
                               (quantifiers
                                 ;; fix NO
                                 (if (member quantifier '(every each))
                                     'singular
                                     'plural))))
    c))

(defvar *numeric-quantifiers*
        (vector (word-for-string "no")
                (word-for-string "one")
                (word-for-string "two")
                (word-for-string "three")
                (word-for-string "four")
                (word-for-string "five")
                (word-for-string "six")
                (word-for-string "seven")
                (word-for-string "eight")
                (word-for-string "nine")
                (word-for-string "ten"))
  "An array for looking up the corresponding determiner for a numeric quantifier")

(defun word-for-quantifier (quantifier)
  (check-type quantifier quantifier)
  (let ((q   quantifier))
    (etypecase q
      (integer     (if (array-in-bounds-p *numeric-quantifiers* q)
                       (aref *numeric-quantifiers* q)
                       (error "no word is defined for quantifier ~s" q)))
      (quantifiers (word-for-string (string-downcase (symbol-name q)))))))
```

```
(defun express-predicate (sexp)
  (let ((verb-concept   (NKGetNamedConcept-error (car sexp)))
        (actuals        (mapcar #'express-formula (cdr sexp))))
    (let ((verb        (word-mapping verb-concept t))
          (rclass-name (rclass-mapping verb-concept t)))
      (let* ((kernel      (apply #'make-a-kernel-with-keywords
                                 rclass-name sexp :verb verb actuals))
             (proposition (clause kernel)))
        (add-accessory proposition :unmarked)
        (add-accessory proposition :tense-modal 'present)
        proposition))))

;;; ============================================================================
;;; Individuals

;;; The NIKL Taxonomy is just for defining terms (a Terminology-Box or T-Box,
;;; see the Krypton article, Brachman, Fikes and Levesque, 1983), we want to
;;; denote individuals with different structures (an Assertion-Box or A-Box).
;;; This file is a trial implementation of this notion.  It just gives us
;;; structures that refer to NIKL concepts and use roles as slots.

;;; 'Slots' is an a-list of the symbolic name of a role permitted by the concept
;;; and its filler, the symbolic name of another nikl-concept.

(defstruct individual
  (concept nil :type (or null nikl-concept))
  (name    nil :type (or null word))
  slots)

(defun create-individual (concept-name proper-name &rest plist-of-roles-and-values)
  (make-individual :concept (NKGetNamedConcept-error concept-name)
                   :name    (the word proper-name)
                   :slots   (plist->alist plist-of-roles-and-values
                                          #'NKGetNamedRole-error
                                          #'NKGetNamedConcept-error)))

(defun express-individual (i)
  (check-type i individual)
  (let ((c  (common-np (word-for-individual i))))
    (link-to-underlying-object c i)
    (add-gender c i)
    (add-accessory c :person 'third)
    (add-accessory c :determiner-policy 'kind)
    (add-accessory c :number 'singular)
    c))

(defun add-gender (bundle item)
  "Uses the 'sex' role of ITEM (a concept or an individual) to set the gender
accessory in BUNDLE (a bundle-specification)"
  (let ((sex    (NKGetNamedRole-error    'sex))
        (male   (NKGetNamedConcept-error 'male))
```

```
        (female (NKGetNamedConcept-error 'female)))
    (let ((vr  (etypecase item
                 (nikl-concept (NKFindVR item sex))
                 (individual   (cdr (assoc sex (individual-slots item)))))))
    (cond ((eq vr male)   (add-accessory bundle :gender 'masculine))
          ((eq vr female) (add-accessory bundle :gender 'feminine))
          (t              (add-accessory bundle :gender 'neuter))))))

;;; When there is no word for this concept, we have to look at
;;; neighboring concepts.  Conjecture:  if there is no word for a
;;; concept concrete enough to have an individual, there will be no
;;; words for any more specific concepts, so we need only look up the
;;; tree.  For example, there's no word for 'wooden furniture,' but
;;; that's a fairly abstract concept, and it's unlikely that you would
;;; have an instance of that concept unless you had no information with
;;; which to make it an instance of a more specific concept.  On the
;;; other hand, consider a concept like 'white tiger': since this is a
;;; fairly specific concept and there is no word for it, it's unlikely
;;; that there would be a word for any more specific concept, such as
;;; 'black female tiger.'  However, for other concepts, it would be
;;; possible to fold in more information.  Consider 'brown horse'; a
;;; more specific concept is 'brown female horse,' but that can be
;;; expressed as 'brown mare.'  It seems that this requires nothing less
;;; than a search of the power set of the individual's features.  But,
;;; we can restrict that search in several ways.  First, we can annotate
;;; roles based on whether they can (in the current lexicon) be folded
;;; into the head noun.  For a role like hair-color, the annotation
;;; would be 'yes,' while for eye-color it would be 'no' (see the main
;;; text).  The annotation is added as local data of a role, under the
;;; key 'foldable?'.  Second, we could rank the features, and only try
;;; to express feature i if features 1 through i-1 are all expressable
;;; (or at least exist as concepts).  This restriction is arguable,
;;; either way.  It does have the advantage of cutting down the search
;;; complexity to linear.  If this restriction is cut out, then the
;;; Rhombus Rule loses some of its force, as we would have to search in
;;; several directions anyhow.  Third, we could consider only subsets of
;;; size less than, say, four.  This seems reasonable, as I cannot think
;;; of any words that incorporate more features, and the restriction
;;; further cuts down the search space to a constant.

;;; Ultimately, the truth of my conjecture depends on the classification
;;; policies of the program.  For now, I will assume that if there is no word
;;; for a concept, we only search upwards (seeking a word for a more general
;;; concept), not downwards (seeking a word for a more specific concept).

(defun word-for-individual (i)
  (check-type i individual)
  (let* ((c   (individual-concept i))
         (w1  (word-mapping c)))
    (cond ((null w1)
           (word-mapping-for-random-superc c))
```

```
          ((decision-point 'search-for-more-specific-word-anyway? i w1)
           (find-more-specific-word i c))
          (T w1))))

(defparameter *max-number-of-features-to-express* 3
  "Conceivably, if an individual has N features, there is a word that expresses
all of the features in one shot.  However, that's pretty unlikely; most words
just express a feature or two, such as sex and age, as with the word ``girl.''
Therefore, to save a little search time, we cut off at this point.")

;;; By assuming the Rhombus Rule, we can search each dimension (role) in
;;; any order we like.  Therefore, we pick a preferred order, so that if
;;; we cannot express the conjunction of features that we want, we can
;;; fall back to previously considered concepts.  That, plus deciding to
;;; try to express fewer than four features (roles), we get the
;;; following simple algorithm.
;;;
;;; We start with CONCEPT and look at all its subconcepts, choosing the
;;; one that fixes the value of a role in the right way.  Then we
;;; iterate, looking at the subconcepts of that concept, and fixing the
;;; value of another role.  When we're done (either no subconcept fixes
;;; the role, or we've done enough features), we take the most recent
;;; concept that has an associated word, and we return that word.

(defun find-more-specific-word (individual concept)
  "We'd like to express some more of INDIVIDUAL's features, if possible.  (A
feature is a dotted pair of a role and its value-restriction.)  Therefore, we
search for a subconcept of CONCEPT such that it expresses all those features
and there is a word for the concept."
  (let* ((all-features   (individual-slots individual))
         (foldable-ones  (remove-if-not #'foldable? all-features :key #'car))
         (features       (preferred-order foldable-ones)))
    (labels ((entails? (test-concept feature)
               "tests if TEST-CONCEPT entails that FEATURE"
               (let ((role  (car feature))
                     (vr    (cdr feature)))
                 (eq (NKFindVR test-concept role) vr)))

             (LP (test-concept best-word remaining-features features-to-go)
               (if (or (null test-concept)
                       (null remaining-features)
                       (zerop features-to-go))
                   best-word
                   (let* ((f      (car remaining-features))
                          (next-c (some #'(lambda (x)
                                            (if (entails? x f) x))
                                        (NKImmediateSubCs test-concept))))
                     (LP next-c
                         (or (word-mapping next-c) best-word)
                         (cdr remaining-features)
                         (- features-to-go 1)))))))
```

```
            )
      (the word
           (LP concept
               (word-mapping concept)
               features
               *max-number-of-features-to-express*)))))

(defun preferred-order (feature-list)
  "Sorts the feature-list into a preferred order of expression.  That is, given
two features, A and B, which would we rather express.  We'll express both if we
can, but which do we retreat to, if A&B cannot be expressed.  By sorting the
features, we need only traverse a single path in the search graph, which is a
computational advantage.  There is, of course, an expressive disadvantage.
Consider if we prefer B to A.  Then we have three options:  1) express A&B, 2)
failing that, express B, and 3) failing that, express neither.

Clearly, this is an arguable algorithm.  I'm not sure I like it myself, but it
is not crazy, and it does make the programming easier.

This function implements the preferred-order rule that sex is the most important
feature to express; otherwise, it doesn't care.  (This demonstrates that the
ranking need not be a total order.)"
  (let ((sex   (NKGetNamedRole-error 'sex)))
    (sort feature-list
          #'(lambda (f1 f2)
              (declare (ignore f2))
              (eq (car f1) sex)))))

;;; ============================================================================
;;; Some special EXPRESS functions

(defun express-simple-subsumption (concept superc)
  (check-type concept nikl-concept)
  (check-type superc  nikl-concept)
  (let ((c1    (if (word-mapping concept)
                   (express-concept concept)
                   (express-concept-with-pronoun concept)))
        (c2    (express-concept superc :do-number? nil)))
    (make-number-agree c1 c2)
    (setq c2 (express-concept-differences concept superc c1 c2))
    (let ((proposition    (clause (make-a-kernel 'predication_to-be c1 c2))))
      (add-accessory proposition :unmarked)
      (add-accessory proposition :tense-modal 'present)
      proposition)))

(defun express-concept-with-pronoun (concept)
  (let* ((plural?    (decision-point 'plural? concept))
         (head       (if plural?
                         (pronoun-named 'these)
                         (pronoun-named 'this)))
         (bundle     (common-np head)))
```

```
    (add-accessory bundle :gender 'neuter)
    (add-accessory bundle :person 'third)
    (add-accessory bundle :number (if plural? 'plural 'singular))
    (add-accessory bundle :determiner-policy 'no-determiner)
    bundle))

(defvar *theta-role* (NKGetNamedRole 'theta)
  "Any theta role of a verb concept will differentiate (in the sense of
NKDifferentiates?) this role.")

(defun theta-role? (role)
  (check-type role nikl-role)
  (NKDifferentiates? role *theta-role*))

(defun express-concept-differences (concept superc concept-msg superc-msg)
  "When we describe a concept with respect to a superconcept, we want also to
express the differences.  For example, ``An Indian Elephant is an Elephant that
comes from India.''  In order to express the differences, we must first know
what they are.  Alas, even though most KR systems know what the differences
are, we are unable to ask them, since that functionality doesn't exist.  This
function is meant to compute the properties that CONCEPT has which SUPER does
not--because of the way we structure our network, these are the reason it's a
subconcept, so we can confidently use restrictive modification.  So far, the
best this function can do is look for role differences."
  (check-type concept nikl-concept)
  (check-type superc  nikl-concept)
  (check-type concept-msg (or bundle-specification pronoun))
  (check-type superc-msg  bundle-specification)
  (let ((diffs   (set-difference (NKFindAllRoles concept)
                                 (NKFindAllRoles superc))))
    (case (length diffs)
      (0  ;; no role differences between them; difference must be primitive
          (a-kind-of concept concept-msg super-msg))
      (1  (cond ((not (theta-role? (car diffs)))
                 (a-kind-of concept concept-msg superc-msg))
                ;; since the diff is a theta-role, either one end or the other
                ;; is a verb-concept.
                ((verb-concept? concept)
                 (error "Can't do nominalizations."))
                (T
                 ;; Since it's not CONCEPT, it must be the other end.
                 (let ((verb-concept  (NKFindVR concept (car diffs))))
                   (add-restrictive-relative-clause
                     concept superc-msg verb-concept)))))
      (T  (error "Can't do multiple restrictions yet.  "
                 "%Just need to do conjunction, but it's complicated."))))

(defun a-kind-of (concept concept-msg superc-msg)
  (check-type concept     nikl-concept)
  (check-type concept-msg bundle-specification)
  (check-type superc-msg  bundle-specification)
```

```
  (if (decision-point 'express-catagory-word?)
      (then (let* ((sw  (specialization-word concept))
                   (o   (np (make-a-kernel 'ako superc-msg sw))))
              (add-accessory o :gender 'neuter)
              (add-accessory o :person 'third)
              (add-accessory o :number 'singular)
              (add-accessory o :determiner-policy 'kind)
              ;; Internally, accessory lists are a-lists, so the last value given
              ;; will win.  Therefore, the following will override express-concept
              (add-accessory superc-msg :number 'singular)
              (add-accessory superc-msg :determiner-policy 'no-determiner)
              o))
      (else (make-number-agree concept-msg superc-msg)
            superc-msg)))

(defun add-restrictive-relative-clause (arg-concept superc-message verb-concept)
  "ARG-CONCEPT is a specialization of its superc because it is an argument of
VERB-CONCEPT.  We express this kind of specialization as a restrictive relative
clause on the expression of its superc, SUPERC-MESSAGE.  For example, 'elephants
that like people' expresses a subconcept of Elephants as 'elephants' plus a
restrictive relative clause."
  (let ((theta-grid  (remove-if-not #'theta-role? (NKFindAllRoles verb-concept)))
        (rclass      (rclass-mapping verb-concept))
        (verb        (word-mapping-for-random-superc verb-concept)))
    (flet ((get-arg (formal)
             ;; FORMAL is actually a Mumble 'parameter' object
             (let ((role-name   (intern (symbol-name (name formal)) 'bitser)))
               (or (NKFindVR verb-concept (NKGetNamedRole-error role-name))
                   (error "~s has no ~s role" verb-concept formal))))

           (make-message (concept)
             (if (eq concept arg-concept)
                 superc-message
                 (express-concept concept)))
           )
      (let* ((formals        (cdr (parameters rclass)))    ;skips VERB parameter
             (arg-concepts   (mapcar #'get-arg formals))
             (arg-messages   (mapcar #'make-message arg-concepts))
             (kernel         (apply #'make-a-kernel rclass verb arg-messages))
             (bundle         (clause kernel)))
        (add-accessory bundle :unmarked)
        (add-accessory bundle :tense-modal 'present)
        (add-specializing-descriptor superc-message bundle)
        superc-message))))

;;; ============================================================================

(defun describe-concept (concept &optional super)
  "Describes a concept in terms of its relationship to a more general concept.
The default is to describe it with respect to its first parent, but any
subsuming concept will do.  Examples:
```

```
   'A rat is a rodent.'
   'A rat is a mammal.'
   'A man is a male human.' (NYI)"
  (check-type concept nikl-concept)
  (check-type super   (or null nikl-concept))
  (let* ((c    concept)
         (s    (or super (car (NKImmediateSuperCs c)))))
    (cond ((parent-concept? s c)
           ;; Is there any difference between describing a concept with respect
           ;; to a parent versus any ancestor?  I can't think of one.
           (express-simple-subsumption c s))
          ((NKSuperC? s c)
           (express-simple-subsumption c s))
          (T
           (error "~s does not subsume ~s" s c)))))

(defun describe-individual (i)
  (check-type i individual)
  (let ((np1  (proper-np (or (individual-name i) (word-for-string "it"))))
        (np2  (express-individual i)))
    (add-accessory np1 :number 'singular)
    (add-accessory np1 :person 'third)
    (add-accessory np1 :determiner-policy 'no-determiner)
    (add-gender np1 i)
    (let ((proposition  (clause (make-a-kernel 'predication_to-be np1 np2))))
      (add-accessory proposition :unmarked)
      (add-accessory proposition :tense-modal 'present)
      proposition)))
```

Appendix B

Demonstration

```
;;; This is a dribble file of a demonstration session with Bitser

(in-package 'bitser)
#<Package BITSER 60466402>

;;; First, look at the KB file to see the toy knowledge base, and the
;;; DATA file to see the annotations on it (local data on concepts).
;;; The following just gets a concept for us to play with.

(setq elephants (NKGetNamedConcept 'elephants))
#<Concept: ELEPHANTS>

;;; DESCRIBE-CONCEPT is a Bitser function which describes a concept with respect
;;; to its superconcept.  It makes use of the word-mapping for each concept.
;;; Note that there are decision points which ask the user for decisions.

(setq m (describe-concept elephants))
Should #<Concept: ELEPHANTS> be plural? (Y or N) Yes.
Express catagory word? (Y or N) No.
#<BUNDLE-SPECIFICATION GENERAL-CLAUSE for PREDICATION_TO-BE>

;;; DISCOURSE-UNIT is a simple function to put a clause bundle into a
;;; discourse-unit bundle, so that Mumble can make a sentence out of it.  Then
;;; MUMBLE-INTO-STRING calls Mumble to generate the sentence and return it.

(mumble-into-string (discourse-unit m))
"Elephants are mammals."

;;; RING-CHANGES is a macro in the decision point code.  It executes its
;;; argument as many times as necessary in order to explore every possible set
;;; of decisions.  Essentially, the decision points define a descrimination tree
;;; and this explores the whole tree.  It prints the decisions and the results,
;;; and returns the whole set of results.

(ring-changes (mumble-into-string (discourse-unit (describe-concept elephants))))
Should #<Concept: ELEPHANTS> be plural?  =>  NIL
```

```
Express catagory word?  =>  NIL

"An elephant is a mammal."

Should #<Concept: ELEPHANTS> be plural?  =>  NIL
Express catagory word?  =>  T
Use specialization word #<WORD SPECIES>?  =>  NIL

"An elephant is a kind of mammal."

Should #<Concept: ELEPHANTS> be plural?  =>  NIL
Express catagory word?  =>  T
Use specialization word #<WORD SPECIES>?  =>  T

"An elephant is a species of mammal."

Should #<Concept: ELEPHANTS> be plural?  =>  T
Express catagory word?  =>  NIL

"Elephants are mammals."

Should #<Concept: ELEPHANTS> be plural?  =>  T
Express catagory word?  =>  T
Use specialization word #<WORD SPECIES>?  =>  NIL

"Elephants are a kind of mammal."

Should #<Concept: ELEPHANTS> be plural?  =>  T
Express catagory word?  =>  T
Use specialization word #<WORD SPECIES>?  =>  T

"Elephants are a species of mammal."

("An elephant is a mammal."
 "An elephant is a kind of mammal."
 "An elephant is a species of mammal."
 "Elephants are mammals."
 "Elephants are a kind of mammal."
 "Elephants are a species of mammal.")


;;; The following just gets our toy set of rules (see the DATA file).

(setq rules (get-concept-rules elephants))
(#S(RULE :QUANTIFIER NO
         :SORT #<Concept: ELEPHANTS>
         :VARIABLE E
         :EXPRESSION (ADMIRE :AGENT E :PATIENT PERSON))
 #S(RULE :QUANTIFIER FEW
         :SORT #<Concept: ELEPHANTS>
         :VARIABLE E
```

```
         :EXPRESSION (DETEST :AGENT E :PATIENT PERSON))
 #S(RULE :QUANTIFIER SOME
         :SORT #<Concept: ELEPHANTS>
         :VARIABLE E
         :EXPRESSION (DISLIKE :AGENT E :PATIENT PERSON))
 #S(RULE :QUANTIFIER MOST
         :SORT #<Concept: ELEPHANTS>
         :VARIABLE E
         :EXPRESSION (LIKE :AGENT E :PATIENT PERSON))
 #S(RULE :QUANTIFIER NORMAL
         :SORT #<Concept: ELEPHANTS>
         :VARIABLE E
         :EXPRESSION (HAVE :AGENT E :PATIENT TUSKS))
 #S(RULE :QUANTIFIER EVERY
         :SORT #<Concept: ELEPHANTS>
         :VARIABLE E
         :EXPRESSION (EAT :AGENT E :PATIENT GRASS)))

;;; Let's just pick one to go through right now.

(setq rule1 (car rules))
#S(RULE :QUANTIFIER NO
        :SORT #<Concept: ELEPHANTS>
        :VARIABLE E
        :EXPRESSION (ADMIRE :AGENT E :PATIENT PERSON))

;;; EXPRESS-FORMULA is a Bitser function to generate a Mumble expression from a
;;; Rule structure.  Look at the function EXPRESS-QUANTIFIED-SORT in the BITSER
;;; file to see how the quantifier determines the number of the subject.  (The
;;; decision point is asking thenumber of the object.)  Also, look at the
;;; function EXPRESS-CONCEPT in the BITSER file to see how distinguishing
;;; between Sorts and Substances decides the determiner-policy of the message.

(setq m (express-formula rule1))
Should #<Concept: PERSON> be plural? (Y or N) Yes.
#<BUNDLE-SPECIFICATION GENERAL-CLAUSE for TRANSITIVE-VERB_TWO-EXPLICIT-ARGS>

;;; Once again, we call Mumble to generate the sentence.

(mumble-into-string (discourse-unit m))
"No elephants admire people."

;;; Now, let's go through all of the rules.  The answers to these decision
;;; points were basically chosen at random.

;;; There are a variety of verbs used, but all using the same realization class.
;;; Look at the DATA file to see the annotations and their inheritance.  See the
;;; note in there about why inheritance of realization classes was probably a
;;; bad idea.

(setq messages (mapcar #'express-formula rules))
```

```
Should #<Concept: PERSON> be plural? (Y or N) Yes.
Should #<Concept: PERSON> be plural? (Y or N) No.
Should #<Concept: PERSON> be plural? (Y or N) Yes.
Should #<Concept: PERSON> be plural? (Y or N) No.
Should #<Concept: TUSKS> be plural? (Y or N) Yes.
(#<BUNDLE-SPECIFICATION GENERAL-CLAUSE for TRANSITIVE-VERB_TWO-EXPLICIT-ARGS>
 #<BUNDLE-SPECIFICATION GENERAL-CLAUSE for TRANSITIVE-VERB_TWO-EXPLICIT-ARGS>
 #<BUNDLE-SPECIFICATION GENERAL-CLAUSE for TRANSITIVE-VERB_TWO-EXPLICIT-ARGS>
 #<BUNDLE-SPECIFICATION GENERAL-CLAUSE for TRANSITIVE-VERB_TWO-EXPLICIT-ARGS>
 #<BUNDLE-SPECIFICATION GENERAL-CLAUSE for TRANSITIVE-VERB_TWO-EXPLICIT-ARGS>
 #<BUNDLE-SPECIFICATION GENERAL-CLAUSE for TRANSITIVE-VERB_TWO-EXPLICIT-ARGS>)

(mapcar #'mumble-into-string (mapcar #'discourse-unit messages))
("No elephants admire people."
 "Few elephants detest a person."
 "Some elephants dislike people."
 "Most elephants like a person."
 "Normal elephants have tusks."
 "Every elephant eats grass.")

;;; The following examples demonstrate expressing Individuals, as opposed to
;;; Concepts.  First, we make a simple individual.

(setq clyde (create-individual 'elephants
                               (word-for-string "Clyde")))
#S(INDIVIDUAL :CONCEPT #<Concept: ELEPHANTS>
              :NAME #<WORD CLYDE>
              :SLOTS NIL)

;;; DESCRIBE-INDIVIDUAL is a Bitser function that describes an individual by
;;; saying what type it is.  Note the decision point which allows the code to
;;; try to be more specific, trying to express the features the individual has.

(mumble-into-string (discourse-unit (describe-individual clyde)))
#S(INDIVIDUAL :CONCEPT #<Concept: ELEPHANTS> :NAME #<WORD CLYDE> :SLOTS NIL)
already has the word #<WORD ELEPHANT>.  Search for a better one anyway? (Y or N) No.
"Clyde is an elephant."

;;; Since Clyde has no additional features, it doesn't matter if we say Yes or No.

(mumble-into-string (discourse-unit (describe-individual clyde)))
#S(INDIVIDUAL :CONCEPT #<Concept: ELEPHANTS> :NAME #<WORD CLYDE> :SLOTS NIL)
already has the word #<WORD ELEPHANT>.  Search for a better one anyway? (Y or N) Yes.
"Clyde is an elephant."

;;; The word-mapping for Elephants was not just a word, but rather a function
;;; that could look at its environment, such as rhetorical goals, and make a
;;; different decision if desired.  Here we have an example.  (The fact that
;;; Elephants maps to a different word doesn't affect whether the program tries
;;; to find a more specific word, and so we still get the decision point query.)
```

```
(let ((*ponderous-pedantic* t))
  (declare (special *ponderous-pedantic*))
  (mumble-into-string (discourse-unit (describe-individual clyde))))

#S(INDIVIDUAL :CONCEPT #<Concept: ELEPHANTS> :NAME #<WORD CLYDE> :SLOTS NIL)
already has the word #<WORD PACHYDERM>.  Search for a better one anyway? (Y or N) No.
"Clyde is a pachyderm."

;;; Now, we create a slightly more complex individual.  Stephanie has one
;;; feature, the fact that her hair is red.  (A 'feature' is a combination of a
;;; Role and a value.  This can be thought of as a Slot and its value, in Frame
;;; terminology.)

(setq stephanie (create-individual 'woman
                                   (word-for-string "Stephanie")
                                   'hair-color 'red))
#S(INDIVIDUAL :CONCEPT #<Concept: WOMAN>
              :NAME #<WORD STEPHANIE>
              :SLOTS ((#<Role: HAIR-COLOR> . #<Concept: RED>)))

;;; The first time, we won't search for a more specific word.

(mumble-into-string (discourse-unit (describe-individual stephanie)))
#S(INDIVIDUAL :CONCEPT #<Concept: WOMAN> :NAME #<WORD STEPHANIE>
:SLOTS ((#<Role: HAIR-COLOR> . #<Concept: RED>))) already has the word #<WORD WOMAN>.
Search for a better one anyway? (Y or N) No.
"Stephanie is a woman."

;;; This time, we will.

(mumble-into-string (discourse-unit (describe-individual stephanie)))
#S(INDIVIDUAL :CONCEPT #<Concept: WOMAN> :NAME #<WORD STEPHANIE>
:SLOTS ((#<Role: HAIR-COLOR> . #<Concept: RED>))) already has the word #<WORD WOMAN>.
Search for a better one anyway? (Y or N) Yes.
"Stephanie is a redhead."

;;; Finally, we consider trying to incorporate two features into the word.
;;; Making this search easier is the motivation for the Rhombus rule.

(setq chris (create-individual 'person
                               (word-for-string "Chris")
                               'sex 'female
                               'occupation 'comedy))
#S(INDIVIDUAL :CONCEPT #<Concept: PERSON>
              :NAME #<WORD CHRIS>
              :SLOTS ((#<Role: SEX> . #<Concept: FEMALE>)
                      (#<Role: OCCUPATION> . #<Concept: COMEDY>)))

(mumble-into-string (discourse-unit (describe-individual chris)))
#S(INDIVIDUAL :CONCEPT #<Concept: PERSON> :NAME #<WORD CHRIS>
:SLOTS ((#<Role: SEX> . #<Concept: FEMALE>) (#<Role: OCCUPATION> . #<Concept:
```

```
COMEDY>)))
already has the word #<WORD PERSON>.  Search for a better one anyway? (Y or N) No.
"Chris is a person."

(mumble-into-string (discourse-unit (describe-individual chris)))
#S(INDIVIDUAL :CONCEPT #<Concept: PERSON> :NAME #<WORD CHRIS>
:SLOTS ((#<Role: SEX> . #<Concept: FEMALE>) (#<Role: OCCUPATION> . #<Concept:
COMEDY>)))
already has the word #<WORD PERSON>.  Search for a better one anyway? (Y or N) Yes.
"Chris is a comedienne."

;;; Our last examples concern expressing concepts for which there is no word.
;;; In this case, we have a subconcept of Elephants, one which differs from that
;;; only in being the agent of liking people.  Thus, we can try to deduce that
;;; difference and construct an expression using the different.  That is, a
;;; restrictive relative clause.

(setq elephants-1 (NKGetNamedConcept 'elephants-that-like-people))
#<Concept: ELEPHANTS-THAT-LIKE-PEOPLE>

;;; Just to remind you there are no tricks up our sleeve:

(word-mapping elephants-1)
NIL

;;; DESCRIBE-CONCEPT is the same function we used above for Elephants.  It uses
;;; the demonstrative pronoun 'these' as the subject because I couldn't think of
;;; anything better.  The point of the sentence is the predicate NP, but Mumble
;;; insists that we generate a sentence.

(mumble-into-string (discourse-unit (describe-concept elephants-1)))
Should #<Concept: ELEPHANTS-THAT-LIKE-PEOPLE> be plural? (Y or N) Yes.
Should #<Concept: PERSON> be plural? (Y or N) Yes.
"These are elephants which like people."

;;; We ring through the changes once again.  Note that some of these sentences
;;; don't sound so great, because the singular sounds like it's an individual
;;; elephant, not a concept.  If I knew what the correct criteria were for
;;; deciding when a concept can and cannot be expressed in the singular, I could
;;; eliminate the decision point.

(ring-changes (mumble-into-string (discourse-unit (describe-concept elephants-1))))
Should #<Concept: ELEPHANTS-THAT-LIKE-PEOPLE> be plural?  =>  NIL
Should #<Concept: PERSON> be plural?  =>  NIL

"This is an elephant which likes a person."

Should #<Concept: ELEPHANTS-THAT-LIKE-PEOPLE> be plural?  =>  NIL
Should #<Concept: PERSON> be plural?  =>  T

"This is an elephant which likes people."
```

```
Should #<Concept: ELEPHANTS-THAT-LIKE-PEOPLE> be plural?  =>  T
Should #<Concept: PERSON> be plural?  =>  NIL

"These are elephants which like a person."

Should #<Concept: ELEPHANTS-THAT-LIKE-PEOPLE> be plural?  =>  T
Should #<Concept: PERSON> be plural?  =>  T

"These are elephants which like people."

("This is an elephant which likes a person." "This is an elephant which likes people."
 "These are elephants which like a person." "These are elephants which like people.")
```

APPENDIX B. DEMONSTRATION

Appendix C

Omissions

The following are examples that appear in the paper but which the program doesn't handle right now. If I return to this work, they should probably be finished.

triangles
polygons
number-of-sides
dangerous elephants
all elephants are mammals
circus elephants normally live a long time
normal elephants have four legs
non-four-legged elephants
albino elephants
normal elephants live in Africa or India
captive elephants
several senators are women
pink elephants
small elephants
poodles
toy poodles
large toy poodles
fat elephants
7,000 pound elephants
toy elephants
African toy elephant
toy African elephant
wooden elephants
free elephants
captive elephants
happy elephants
dying elephants
talented elephants
bettors
employer

employee
employ
elephants that are afraid of mice
elephants that fear mice
Horse: graminivorous, ...