4.900

Prolog Tutorial 4

by Matthew Huntbach

Prolog Tutorial 4

Logic Programming Associates Ltd

Studio 30

The Royal Victoria Patriotic Building

Trinity Road

London SW18 3SX

England

phone: +44 (0) 20 8871 2016

fax: +44 (0) 20 8874 0449

web: http://www.lpa.co.uk

4 February, 2005

Contents

Notes on Semantic Nets and Frames

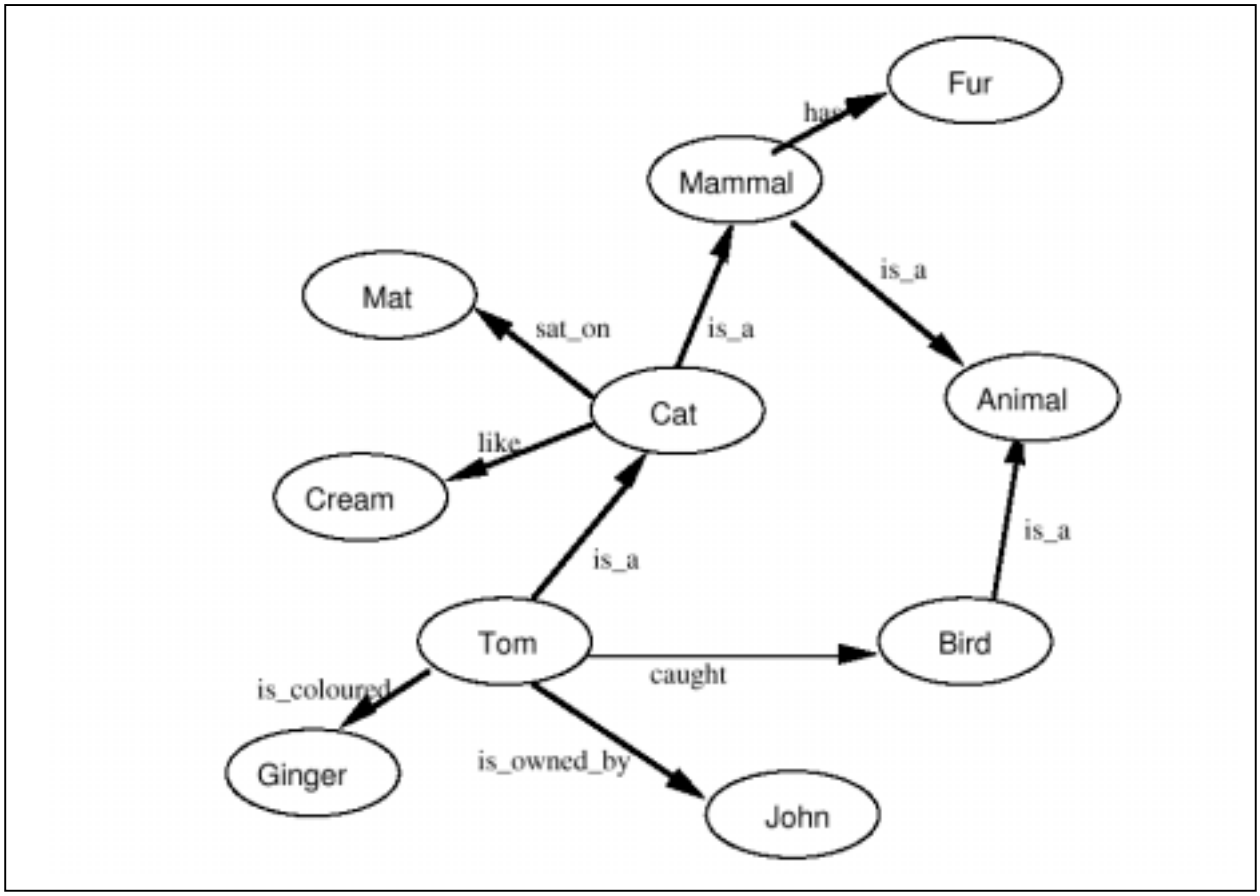
These notes were written by Matthew Huntbach, Dept of Computer Science, Queen Mary and Westfield College, London, UK E1 4NS. Email: mmh@dcs.qmw.ac.uk.

These notes were edited by Clive Spenser, LPA.

These notes show how we can model semantic networks in Prolog and/or Flex.

Semantic Nets

Semantic networks are an alternative to predicate logic as a form of knowledge representation. The idea is that we can store our knowledge in the form of a graph, with nodes representing objects in the world, and arcs representing relationships between those objects. For example, the following:



is intended to represent the data:

Tom is a cat.

Tom caught a bird.

Tom is owned by John.

Tom is ginger in colour.

Cats like cream.

The cat sat on the mat.

A cat is a mammal.

A bird is an animal.

All mammals are animals.

Mammals have fur.

It is argued that this form of representation is closer to the way humans structure knowledge by building mental links between things than the predicate logic we considered earlier. Note in particular how all the information about a particular object is concentrated on the node representing that object, rather than scattered around several clauses in logic.

There is, however, some confusion here which stems from the imprecise nature of semantic nets. A particular problem is that we haven't distinguished between nodes representing classes of things, and nodes representing individual objects. So, for example, the node labelled Cat represents both the single (nameless) cat who sat on the mat, and the whole class of cats to which Tom belongs, which are mammals and which like cream. The is_a link has two different meanings – it can mean that one object is an individual item from a class, for example Tom is a member of the class of cats, or that one class is a subset of another, for example, the class of cats is a subset of the class of mammals. This confusion does not occur in logic, where the use of quantifiers, names and predicates makes it clear what we mean so:

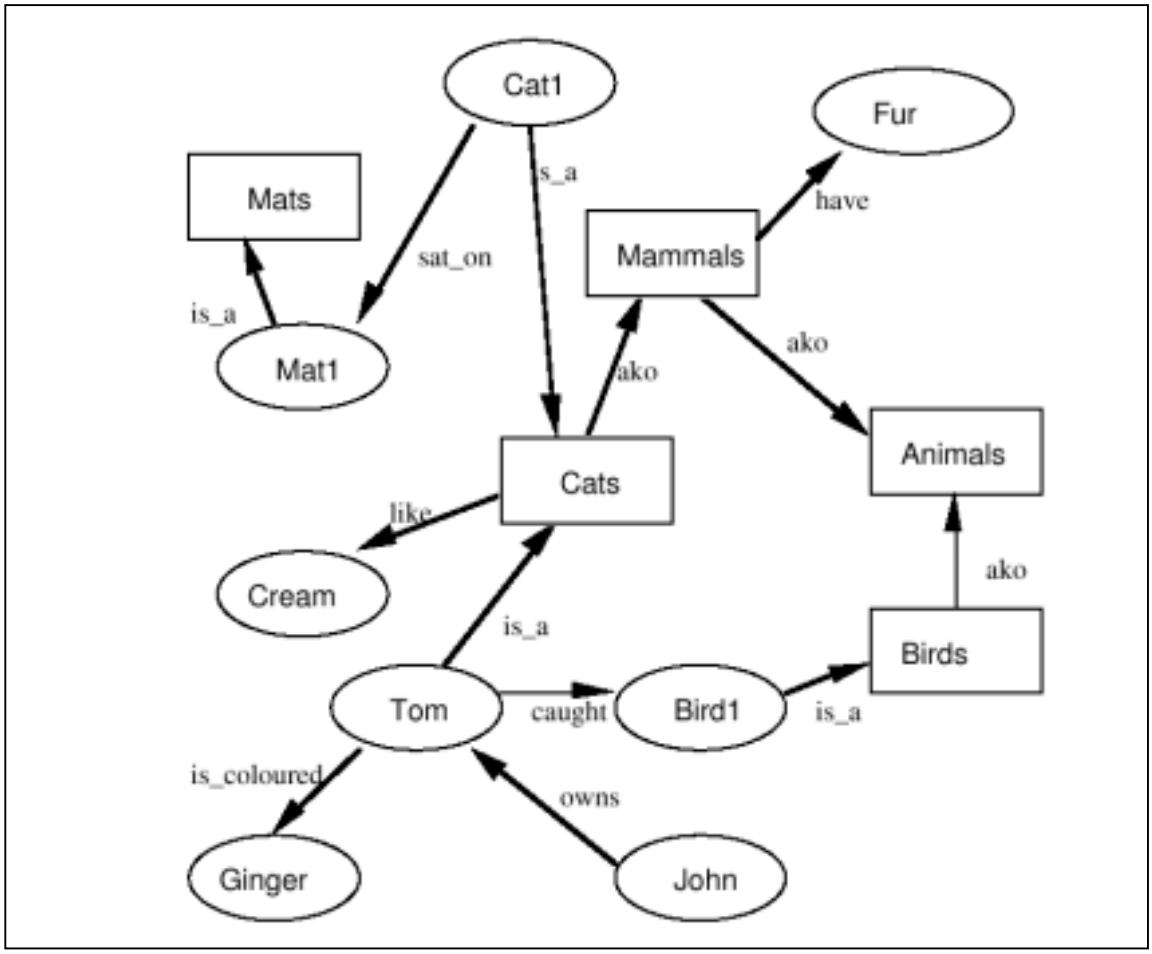
Tom is a cat is represented by Cat(Tom)

The cat sat on the mat is represented by $\exists x \exists y(Cat(x) \wedge Mat(y) \wedge SatOn(x,y))$

A cat is a mammal is represented by $\forall x(Cat(X) \rightarrow Mammal(x))$

We can clean up the representation by distinguishing between nodes representing individual or instances, and nodes representing classes. The is_a link will only be used to show an individual belonging to a class. The link representing one class being a subset of another will be labelled a_kind_of, or ako for short. The names instance and subclass are often used in the place of is_a and ako, but we will use these terms with a slightly different meaning in the section on Frames below.

Note also the modification which causes the link labelled is_owned_by to be reversed in direction. This is in order to avoid links representing passive relationships. In general a passive sentence can be replaced by an active one, so "Tom is owned by John" becomes "John owns Tom". In general the rule which converts passive to active in English converts sentences of the form "X is Yed by Z" to "Z Ys X". This is just an example (though often used for illustration) of the much more general principle of looking beyond the immediate surface structure of a sentence to find its deep structure. The revised semantic net is:


Cat1
Fur
Mats
s_a
have
Mammals
sat_on
is_a
Mat1
ako
ako
Animals
Cats
like
Cream
ako
is_a
Birds
Tom
caught
Bird1
is_a
is_coloured
owns
Ginger
John

Note that where we had an unnamed member of some class, we have had to introduce a node with an invented name to represent a particular member of the class. This is a process similar to the Skolemisation we considered previously as a way of dealing with existential quantifiers. For example, "Tom caught a bird" would be represented in logic by ∃x(bird(x)∧caught(Tom,x)), which would be Skolemised by replacing the x with a Skolem constant; the same thing was done above where bird1 was the name given to the individual bird that Tom caught.

There are still plenty of issues to be resolved if we really want to represent what is meant by the English phrases, or to be really clear about what the semantic net means, but we are getting towards a notation that can be used practically (one example of a thing we have skated over is how to deal with mass nouns like "fur" or "cream" which refer to things that come in amounts rather than individual objects).

A direct Prolog representation can be used, with classes represented by predicates, thus:

```
cat(tom).
cat(cat1).
mat(mat1).
sat_on(cat1,mat1).
bird(bird1).
caught(tom,bird1).
like(X,cream) :-
   cat(X).

mammal(X) :-
   cat(X).

has(X,fur) :-
   mammal(X).

animal(X) :-
   mammal(X).

animal(X) :-
   bird(X).

owns(john,tom).
is_coloured(tom,ginger).
```

So, in general, an is_a link between a class c and an individual m is represented by the fact c(m). An a_kind_of link between a subclass c and a superclass s is represented by s(X) :- c(X). If a property p with further arguments a1, … ,an is held by all members of a class c, it is represented by p(X,a1,…,an) :- c(X). If a property p with further arguments a1, … ,an is specified as held by an individual m, rather than a class to which m belongs, it is represented by p(m,a1,…,an).

In Flex, this could be represented as:

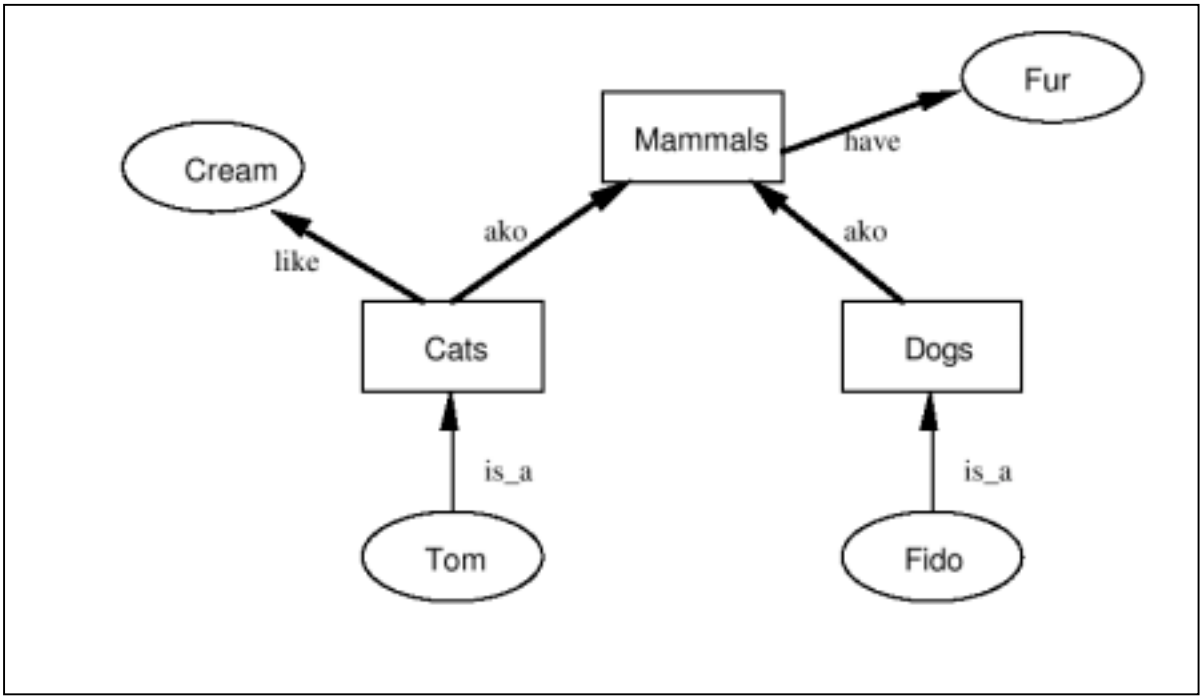
```
frame animal;

frame mammal is a animal;
   default skin is fur.

frame cat is a mammal
   default like is cream.
```

```
instance tom is a cat.
instance cat1 is a cat.

frame mat.

instance mat1 is a mat.

relation sat_on(cat1,mat1) if true.

frame bird is a animal.

instance bird1 is a bird.

relation caught(tom,bird1) if true.

relation owns(john,tom) ) if true.
```

Inheritance

This Prolog equivalent captures an important property of semantic nets, that they may be used for a form of inference known as inheritance. The idea of this is that if an object belongs to a class (indicated by an is_a link) it inherits all the properties of that class. So, for example as we have a likes link between cats and cream, meaning "all cats like cream", we can infer that any object which has an is_a link to cats will like cream. So both Tom and Cat1 like cream. However, the is_coloured link is between Tom and ginger, not between cats and ginger, indicating that being ginger is a property of Tom as an individual, and not of all cats. We cannot say that Cat1 is ginger, for example; if we wanted to we would have to put another is_coloured link between Cat1 and ginger.

Inheritance also applies across the a_kind_of links. For example, any property of mammals or animals will automatically be a property of cats. So we can infer, for example, that Tom has fur, since Tom is a cat, a cat is a kind of mammal, and mammals have fur. If, for example, we had another subclass of mammals, say dogs, and we had, say, Fido is_a dog, Fido would inherit the property has fur from mammals, but not the property likes cream, which is specific to cats. This situation is shown in the diagram below:



Reification

An alternative form of representation considers the semantic network directly as a graph. We have already seen ways of representing graphs in Prolog. We could represent each edge in the semantic net graph by a fact whose predicate name is the label on the edge. The nodes in this graph, whether they represent individuals or classes are represented by arguments to the facts representing edges. This gives the following representation for our initial graph: is_a(mat1,mats).

```
is_a(cat1,cats).
is_a(tom,cats).
is_a(bird1,birds).
caught(tom,bird1).
ako(cats,mammals).
ako(mammals,animals).
ako(birds,animals).
like(cats,cream).
owns(john,tom).
sat_on(cat1,mat1).
is_coloured(tom,ginger).
have(mammals,fur).
```

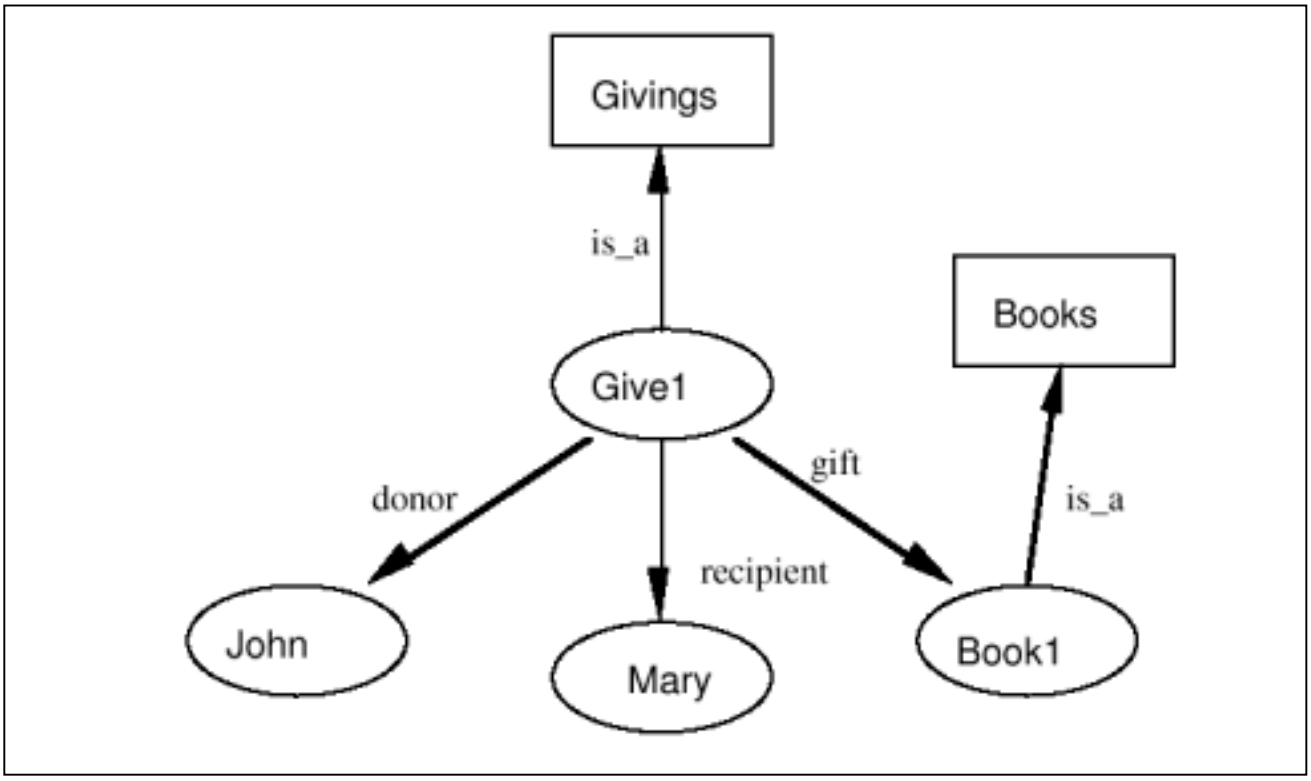
Alternatively, the graph could be built using the cells or pointers of an imperative language. There are also special purpose knowledge representation languages which provide a notation which translates directly to this sort of graph.

This process of turning a predicate into an object in a knowledge representation system is known as reification. So, for example, the constant symbol cats represents the set of all cats, which we can treat as just another object.

The Case for Case

We have shown how binary relationships may be represented by arcs in graphs, but what about relationships with more than two arguments? For example, what about representing the sentence

"John gave the book to Mary"? In predicate logic, we could have a 3-ary predicate gave, whose first argument is the giver, second argument the object given and third argument the person to whom it was given, thus gave(John,Book1,Mary). The way this can be resolved is to consider the act of giving a separate object (remember how in the first set of notes we saw how the pronoun "it" could in some contexts be taken to refer to a previously mentioned action itself rather than to an object involved in the action), thus it is further reification. We can than say that any particular act of giving has three participants: the donor, the recipient, and the gift, so the semantic net representing the sentence is:



In fact the three different roles correspond to what is known in natural language grammar as subjective (the object doing the action, in this case John), objective (the object to which the action is being done, in this case the book) and dative (the recipient of the action, in this case Mary). These different roles of objects in a sentence are known as cases.

The fact that various natural languages make this case distinction can be used to support using it in artificial knowledge representation. The "case for case" is associated with the linguist Charles Fillmore whose work has been influential among AI workers in knowledge representation. The idea is that all sentences can be analysed as an action

plus a number of objects filling the roles in the action, with there being a fixed set of roles (though not every role will always be filled). Other roles suggested as fundamental include the locative indicating where the action is done, and the instrumental, indicating the means by which an action is done.

In some natural languages the different roles which a word may fill are indicated by the ending or inflexion of the word. A well-known example of such an inflexional language is Latin (but some modern languages, such as Russian are equally as inflexional), where, for example "Dog bites man" is "Canis hominem mordet" while "Man bites dog" is "Homo canem mordet". The word for "dog" is "canis" if it is the object of the sentence, but "canem" if it is the subject, while for "man" it is "homo" if he is the object of the sentence and "hominem" if he is the subject. If something were being given to a dog, the word used would be "cane", if a dog were being used for something the word used would be "cani". In English the objective and subjective roles are indicated by word order, with the object coming before the verb and the object coming after. In Latin, it is the case endings, not the word order that indicates a role, so "Hominem canis mordet" is just another way of saying "Dog bites man". You could perhaps compare it the programming languages where the relationship of arguments to formal parameter names in procedure calls may be indicated by their position, but in some cases (e.g. Modula-3) a facility is available for named arguments.

In English the dative is occasionally indicated by word order (for example in "John gave Mary the book"), but more often by prefixing the word indicating the dative item with the preposition "to", as in "John gave the book to Mary". Other cases are always indicated by prepositions, for example the locative with "at" (e.g. "John gave the book to Mary at school") and the instrumental with "by" or "with" ("John sent the book to Mary by post", "Mary hit John with the book"). Most inflexional languages have a limited range of cases, and use prepositions to extend the range. In fact the argument for case as innate is damaged by the fact that different languages have different case structures, and it is by no means certain which cases are fundamental and which are just variants of others. For example, in sentences involving the concept of movement linguists distinguish the ablative case (the source of the movement, in English indicated by the preposition "from") and the allative case (the destination of the movement), but should the latter be considered just another form of the dative role?

Using the concept of a semantic network in which nodes represent individual actions, with arcs representing objects having roles in these actions, it is possible to build up complex graphs representing complete scenarios. For example, the story:

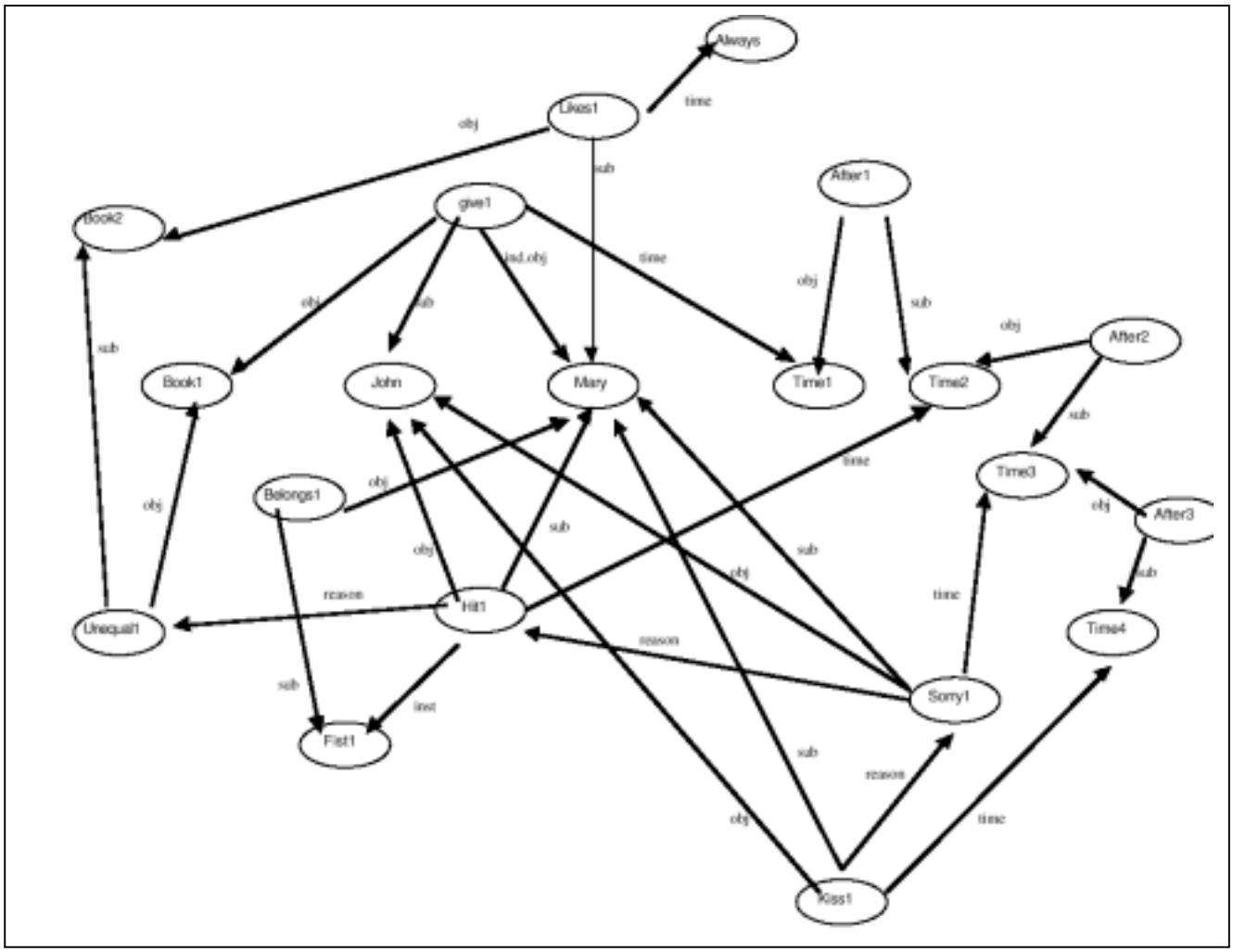
> "John gave Mary a book. The book was not the one Mary likes, so she punched John. That made her feel sorry for him, so she then kissed him"

is represented by the graph on the next page. The class nodes are omitted as the graph is complex enough without them. The arcs are labelled with sub and obj, for the subject and object of the action, ind.obj and instr for the case where there is an indirect object (i.e. dative in the terminology used previously) and an instrument. There are also arcs representing time relationships

– note that individual times are represented by nodes as well, and reasons why an act was performed.

Note that in the graph some English words are translated to an equivalent, thus "punch" is represented as "hit with fist" (we might also, for example, have represented

"kiss" by "touch with lips", though this perhaps illustrates why this sort of attempt to find an underlying representation can miss some of the subtleties of human language!). Similarly, if we are trying to represent underlying meanings, we have not only to convert passive forms to active forms as suggested previously, but also to note forms where one verb is equivalent to another, except with the roles in a different order. For example, the sentence "X buys Y from Z" is essentially equivalent to "Z sells Y to X", so we could therefore convert all sentences involving selling to the equivalent involving buying and make them instances of the buying class. Work on trying to find underlying primitives to aid network representation of the meaning of natural language semantics is associated with the AI researcher Roger Schank.



The information in this graph could be represented by a series of logical facts like the set of Prolog facts we gave as the first representation of the previous graph. The advantage of the graph notation is that it may be more intuitive, and in particular it brings together all the information associated with a particular individual. Drawing inferences from a semantic net involves searching for particular patterns. For example, the question "Who kissed John?" from the above graph involves searching for a node which links to the class node kissings with an is_a link (this is one of the links not shown), and has an object link to the node representing John. The answer to the question is then found from the subject link of that node.

In Prolog this would be the query:

```
is_a(K,kissings), object(K,john), subject(K,Answer). .
```

Note that the graph may represent a scenario where John is kissed more than once, in which case there would be more than one node fitting the conditions, and the query could be made to backtrack to give alternate answers.

In flex it would look something like:

```
check that K is a kissings whose object is john and
check that A is the subject of K .
```

A "whom" question is a search for the object of a node given the subject, thus "Mary kissed whom?" (modern English is more likely to phrase this "Who did Mary kiss?", the distinction between "who" as a query for a subject and "whom" as a query for an object being lost) is represented by:

```
is_a(K,kissings), subject(K,mary), object(K,Answer).

check that K is a kissings whose subject is mary and
check that A is the object of K .
```

Similarly a "to whom" question is a search for an indirect object given a subject and object, so "John gave the book to whom?" or "Who did John give the book to?" is represented by:

```
is_a(G,givings), subject(G,john), object(G,B), is_a(B,book),
indirect_object(B,Answer).

check that G is a givings whose subject is john and
check that whose object is B and
check that B is a book and
check that Answer is the indirect_object of G.
```

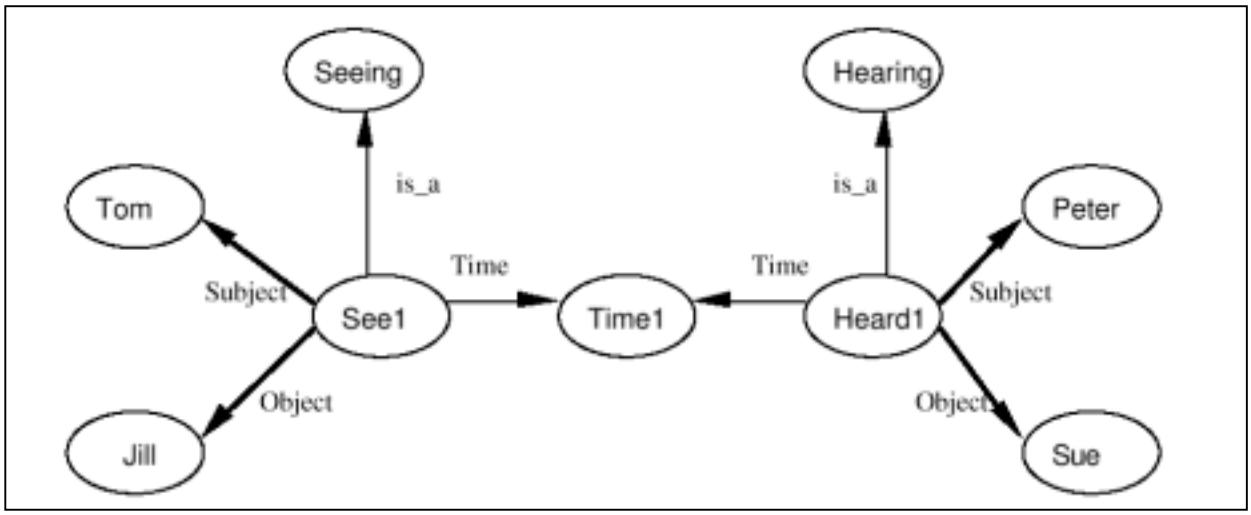
A "how" question might be considered equivalent to a "with what" question, so it is returns the instrumental link of the relevant node. A "where" question returns the locative link.

A "why" question is a search for a reason link, so "Why did Mary kiss John?" is represented by:

```
is_a(K,kissings), subject(K,mary), object(K,john), reason(K,Answer)
```

In this case, however, the answer will not be an individual but simply a name assigned to an node representing a feeling_sorry_for action. A more correct report would need to give the complete sentence represented by the node to which the reason link points.

Similarly, a "when" question is a search for a time link. Time links may point to nodes actually storing times and dates. However, as in our example, it is more likely to be a time which is relative to another, so again the answer given must involve looking beyond just the node pointed to by the time link. For example, with our above graph the question "When did Mary feel sorry for John" would be answered by finding that the time link from the node sorry1 links to time time3. It can then be noted that time3 is the subject of one after node, and the object of another, so the answer could be given as both "After Mary hit John" and "Before Mary kissed John". If two different action nodes pointed to the same time node, the time of the action of one could be given as when the other happened, so for example with the graph below:



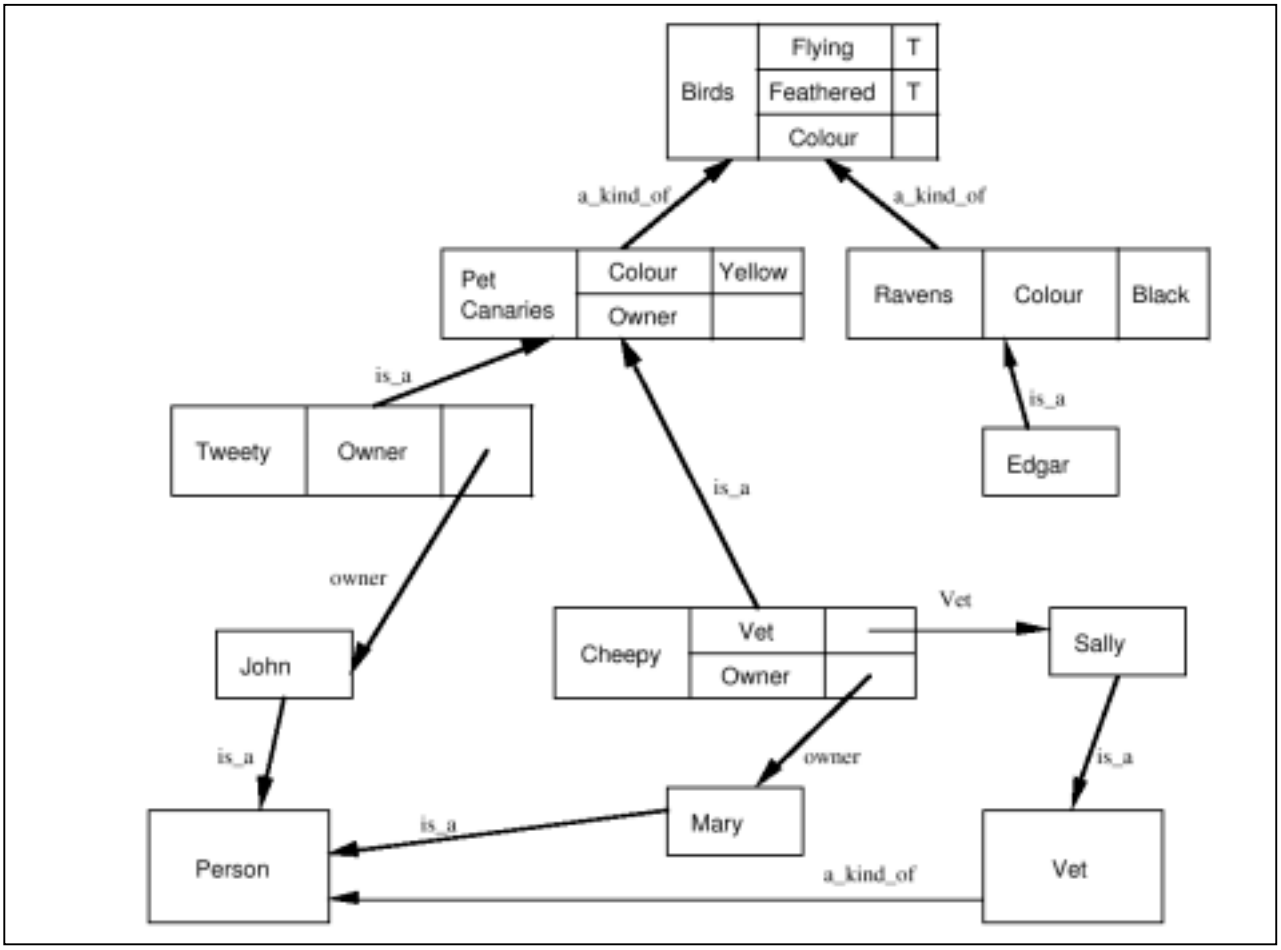
the question "When did Tom see Jill?" could be answered "When Peter heard Sue". Note that a simplification we are making here is that all actions occur instantly at a fixed time point. A more realistic treatment of time would deal with time intervals which have a start and finish time.

Frames, Slots and Fillers

Consideration of the use of cases suggests how we can tighten up on the semantic net notation to give something which is more consistent, known as the frame notation. In the place of an arbitary number of arcs leading from a node there are a fixed number of slots representing attributes of an object. Every object is a member or instance of a class, which it may be thought of as linking to with an is_a link as we saw before. The class indicates the number of slots that an object has, and the name of each slot. In the case of a giving object, for instance, the class of giving objects will indicate that it has at least three slots: the donor, the recipient and the gift. There may be further slots indicated as necessary in the class, such as ones to give the time and location of the action. The time slot may be considered a formalisation of the tense of the verb in a sentence.

The idea of inheritance is used, with some slots being filled at class level, and some at instance level. Where a slot is filled at class level the idea is that this represents attributes which are common to all members of that class. Where it is filled at instance level, it indicates that the value of that attribute varies among members of that class. Slots may be filled with values or with pointers to other objects. This is best illustrated by an example.

In our example we have a general class of birds, and all birds have attributes flying, feathered and colour. The attributes flying and feathered are boolean values and are fixed to true at this level, which means that for all birds the attribute flying is true and the attribute feathered is true. The attribute colour, though defined at this level is not filled, which means that though all birds have a colour, their colour varies. Two subclasses of birds, pet_canaries and ravens are defined. Both have the colour slot filled in, pet_canaries with yellow, ravens with black. The class pet_canaries has an additional slot, owner, meaning that all pet canaries have an owner, though it is not filled at this level since it is obviously not the case that all pet canaries have the same owner. We can therefore say that any instance of the class pet_canary has attributes colour yellow, feathered true, flying true, and owner, the last of these varying among instances. Any instance of class raven has colour black, feathered true, flying true, but no attribute owner. The two instances of pet_canary shown, Tweety and Cheepy have owners John and Mary who are separate instances of the class person, for simplicity no attributes have been given for class person. The instance of pet_canary Cheepy has an attribute which is restricted to itself, vet (since not all pet canaries have their own vet), which is a link to another person instance, but in this case we have subclass of person, vet. The frame diagram for this is:



We can write this in Flex as:

```
frame person.

frame     vet is a person.


instance sally is a vet.

instance john is a person.

instance mary is a person.


frame birds
   default flying is true and
   default feathered is true .

frame pet_canaries is a birds
   default colour is yellow .

frame raven        is a birds
   default colour is black .
```

```
instance cheepy is a pet_canaries ;
   vet is sally and
   owner is mary .

instance tweety is a per_canaries ;
   owner is john .

instance tweety is a raven .
```

We can define a general set of rules for making inferences on this sort of frame system. We can say that an object is an instance of a class if it is a member of that class, or if it is a member of a class which is a subclass of that class. A class is a subclass of another class if it is a kind of that class, or if it is a kind of some other class which is a subclass of that class. An object has a particular attribute if it has that attribute itself, or if it is an instance of a class that has that attribute. In Prolog:

```
aninstance(Obj,Class) :-
   is_a(Obj,Class).

aninstance(Obj,Class) :-
   is_a(Obj,Class1),
   subclass(Class1,Class).

subclass(Class1,Class2) :-
   a_kind_of(Class1,Class2).

subclass(Class1,Class2) :-
   a_kind_of(Class1,Class3),
   subclass(Class3,Class2).
```

We can then say that an object has a property with a particular value if the object itself has an attribute slot with that value, or it is an instance of a class which has an attribute slot with that value, in Prolog:

```
value(Obj,Property,Value) :-
   attribute(Obj,Property,Value).

value(Obj,Property,Value):-
   aninstance(Obj,Class),
   attribute(Class,Property,Value).
```

The previous diagram could then be represented by the following Prolog facts:

```
attribute(birds,flying,true).
attribute(birds,feathered,true). attribute(pet_canaries,colour,yellow).
attribute(ravens,colour,black).
attribute(tweety,owner,john).
attribute(cheepy,owner,mary).
attribute(cheepy,vet,sally).
a_kind_of(pet_canaries,birds).
a_kind_of(ravens,birds).
a_kind_of(vet,person).
is_a(edgar,ravens).
is_a(tweety,pet_canaries).
is_a(cheepy,pet_canaries).
is_a(sally,vet).
```

```
is_a(john,person).
is_a(mary,person).
```

Note in particular how we have used reification leading to a representation of classes like birds, pet_canaries and so on by object constants, rather than by predicates as would be the case if we represented this situation in straightforward predicate logic. The term superclass may also be used, with X being a superclass of Y whenever Y is a subclass of X.

Using the Prolog representation, we can ask various queries about the situation represented by the frame system, for example if we made the Prolog query:

```
?- value(tweety,colour,V).
```

we would get the response:

```
V = yellow ?
```

while

```
?- value(john,feathered,V).
```

gives the response

```
no
```

indicating that feathered is not an attribute of John. Note that the no indicates that this is something which is not recorded in the system. If we wanted to actually store the information that persons are not feathered we would have to add:

```
attribute(person,feathered,true).
```

then the response would have been:

```
V = false ?
```

The only thing that has not been captured in this Prolog representation is the way that an attribute can be defined at one level and filled in lower down, like the colour attribute of birds.

Demons and Object-Oriented Programming

Some frame systems have an additional facility in which a slot may be filled not by a fixed attribute but by a procedure for calculating the value of some attribute. This procedure is known as a demon (the name coming from the idea that it "lurks around waiting to be invoked"). A demon may be attached to a class, but make use of information stored in a subclass or an instance.

For instance, in the above example, we might want to have an attribute maintenance representing maintenance costs attached to the subclass pet_canaries, which should return £5 for a pet canary without its own vet, but £5+vet's fees for a canary with a vet. However, if we do this we will need to have a way to refer to the individual instance of a class at the class level. We do this through the use of a variable conventionally called Self. We then need to add the reference to Self to our rules for determining the value of some property:

```
value(Obj,Property,Value) :-
   attribute(Obj,Obj,Property,Value).

value(Obj,Property,Value) :-
   aninstance(Obj,Class),
   attribute(Obj,Class,Property,Value).
```

The first argument to attribute here is the reference to Self. Our previous attributes do not depend on the value of Self, so we can just add it as an anonymous variable: attribute(_,birds,flying,true).

```
attribute(_,pet_canaries,colour,yellow).
```

and so on for the other attributes. For our example, we must have the attribute fees attached to vets (it will vary from vet to vet so it will be filled in at instance level), so we will also add to our example:

```
attribute(_,sally,fees,20).
```

Now, to add our demon, which we will name eval_maintenance, we add:

```
attribute(Self,pet_canaries,maintenance,Costs) :-
   eval_maintenance(Self,Costs).

eval_maintenance(Self,Costs) :-
   value(Self,vet,SelfsVet),
   !,
   value(SelfsVet,fees,VetFees),
   Costs is VetFees+5.

eval_maintenance(Self,5).
```

The use of the cut here is because the only way we can find out if a pet canary doesn't have a vet is to see if fails, but we don't want backtracking for a pet canary that does have a vet to give an alternative value for maintenance costs.

The introduction of demons brings our knowledge representation method close to that

of object- oriented programming. Several object-oriented programming language have been developed which give mechanisms directly for expressing classes with attached procedures and inheritance. The most successful examples are C++ and Smalltalk. Development of the idea of demons into full procedures which may change the values stored with an object moves away from the declarative ideas of knowledge representation, so we shall not develop it further here, but those taking the course in Object-Oriented Programming will be able to build the connection.

Defaults and Overrides

One of the problems we mentioned with predicate logic is that it does not provide us with a way of saying that some particular conclusion may be drawn unless we can show otherwise. We had to add the idea of negation as failure to deal with this, and even then if we want to draw a conclusion we have to show that all the conditions that would cause that conclusion to fail are false.

For example, we know that in general birds can fly. So we can write in Prolog:

```
flies(X) :- bird(X).
```

But suppose we want to deal with special cases of birds that cannot fly. We know that kiwis and penguins cannot fly, for instance. We also know that any bird with a broken wing cannot fly. So strictly we would have to say:

```
flies(X) :- bird(X), \+kiwi(X), \+penguin(X), \+broken_wing(X).
```

We can summarise this as:

```
flies(X) :-
  bird(X),
  \+ab(X).
```

where ab(X) means "X is an abnormal bird". We could list the factors that make X an abnormal bird in respect to flying:

```
ab(X) :-
  kiwi(X).
ab(X) :-
  penguin(X).

ab(X) :-
  broken_wing(X).
```

but there might always be circumstances we had not thought of (other species of birds that don't fly, birds whose wings are not broken but whose feet are trapped, etc.). As we mentioned in a previous set of notes forms of default logic exist which enable us to say that some conclusion holds on the assumption that there are no facts known to indicate why they should not. So we might say that bird(x) is true with assumption set {¬ab(x)}. This is non-monotonic reasoning, since the addition of a fact which makes some assumption false will make a conclusion false. For example, if we have ostrich(ossie) and bird(X):–ostrich(X) we can assume flies(ossie), but if we add ab(X):–ostrich(X), this reasoning fails. In practice there will have to be a separate form of ab for every rule.

Another way of putting this is to say that the default is for any bird x, flies(x) is true.

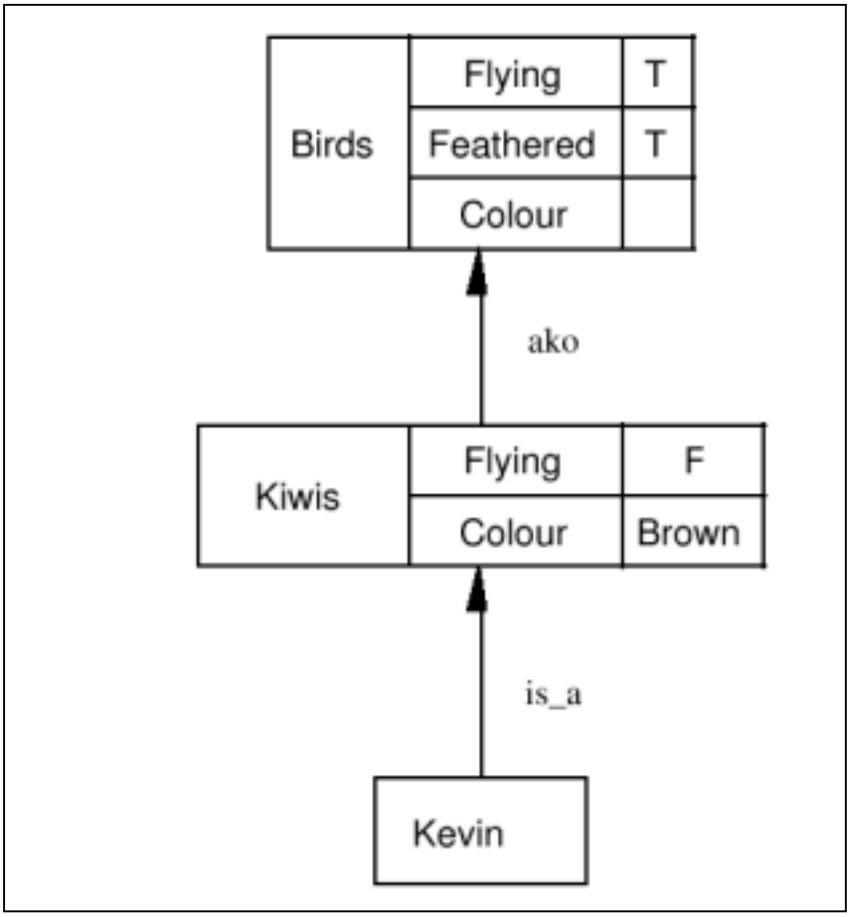
Default reasoning is easily added to the frame system of representation. The idea used is that an attribute at class level is inherited only if it is not cancelled out or overridden by the same attribute slot occurring in a subclass of that class or in an individual instance with a different value. For example, we could add the class of kiwis as a

subclass of birds in our diagram above, and indicate that kiwis cannot fly. The additional attributes to create a class of kiwis with one instance kevin are:

```
a_kind_of(kiwis,birds).
attribute(kiwis,flying,false).
attribute(kiwis,colour,brown).

is_a(kevin,kiwis).
```

We have to add a colour attribute for kiwis as this was a slot in its superclass, birds. For simplicity we have gone back to the representation which does not allow for the possibility of demons.

The following arcs are added to our diagram:



Now it will be seen that for X=tweety, cheepy or edgar,

```
?- value(X,flying,V).
```

will give the response

```
V = true ?
```

but

```
?- value(kevin,flying,V).
```

will give the response

```
V = false ?
```

One problem is that if we typed ; in response to this we would get:

```
V = true ?
```

In order to prevent this possibility we need to put cuts in our inference rules, so that when the property is found it is not possible to backtrack and search higher in the inheritance tree for a value for the same property:

```
value(Obj,Property,Value) :-
attribute(Obj,Property,Value), !.

value(Obj,Property,Value):-
 aninstance(Obj,Class),
attribute(Class,Property,Value), !.
```

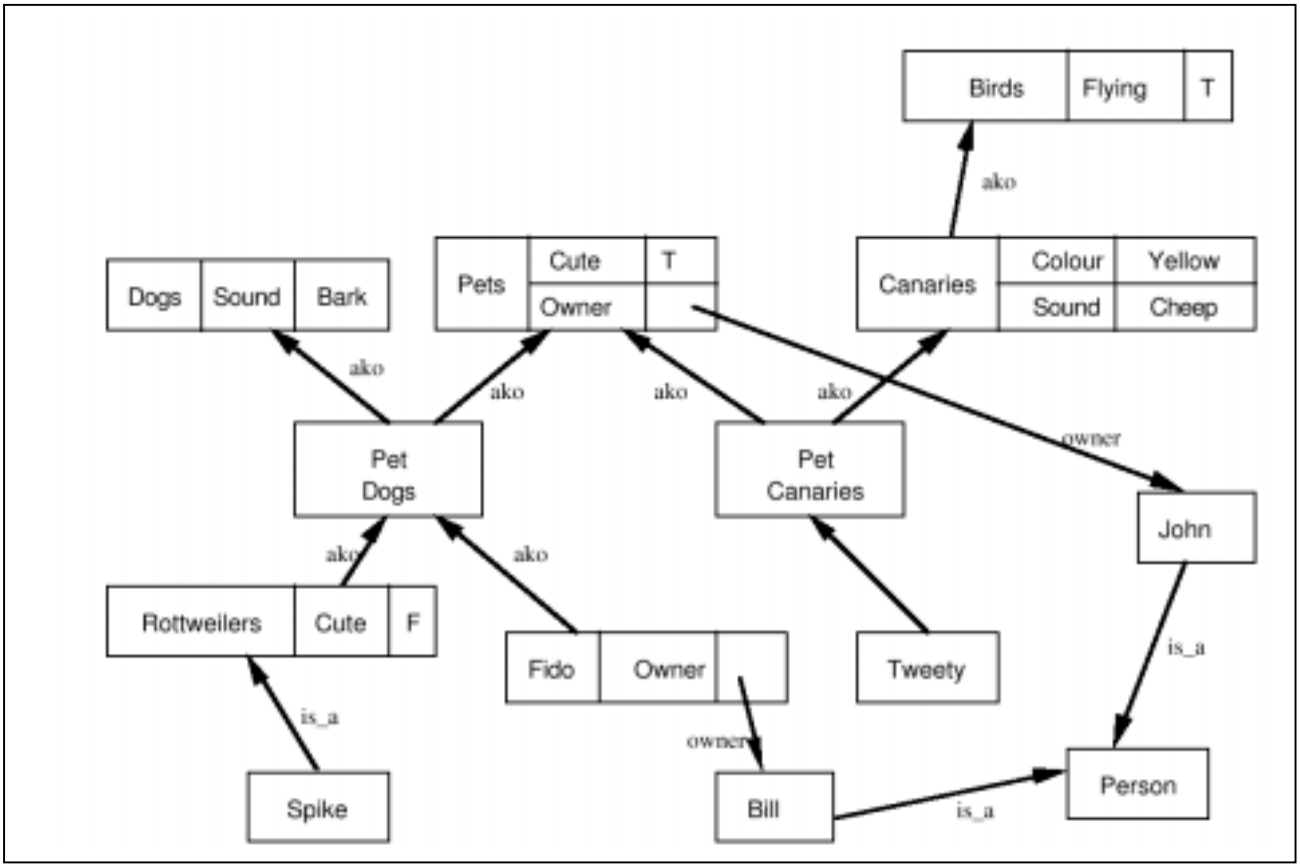
The presence of the cut indicates that we have lost the strict declarative reading, and the result we get will depend on the ordering of the rules. This will become more apparent when we consider multiple inheritance next.

The result of adding the possibility of overrides is that the information stored at class level no longer represents attributes held by all members of that class, but can be taken as being the attributes held by the typical member of that class. Sometimes the class level node in the inheritance tree is said to represent the prototype member of that class. All new instances of that class are constructed by taking the prototype and altering the defaults as required.

In order to establish coherency, sometimes a distinction is made between defining attributes which cannot be overridden, and default attributes which can. Any attempt to add a node to the inheritance graph which overrode a defining attribute would be flagged as an error. Without this feature it would, for example, be possible to define a subclass in which all the attributes of a superclass are overridden.

Multiple Inheritance

We have not said anything that indicates that an object may not be an instance of more than one class, or a class be a subclass of more than one class. In fact this can easily be done within our existing system simply by not insisting that every fact is_a(X,Y) or a_kind_of(X,Y) has a unique value for X. This is described as multiple inheritance . Again, let us consider an example, slightly different from the one above. We will again be representing information about pet canaries, but this time we will have a separate class of pets and a class of canaries. The class of pet canaries inherits properties from both pets and canaries. We will assume that pets have the default property of being cute, birds have the fault property of flying, and canaries the default properties of being coloured yellow, and making the sound cheep. For comparison, we will also add a class of pet dogs. All dogs have the default property that the sound they make is a bark. To illustrate a default being overridden we include the class of Rottweilers, a subclass of pet_dogs where the property that cute is true is overridden by cute being false. The diagram is:



We have also added that John is the default owner of any pet, so any pet whose owner we don't know we assume is John's.

The Prolog facts representing this set up are:

```
attribute(birds,flying,true).
```

```
attribute(dogs,sound,bark).

attribute(pets,cute,true).

attribute(pets,owner,john).

attribute(canaries,colour,yellow).

attribute(canaries,sound,cheep).

attribute(rottweilers,cute,false).

attribute(fido,owner,bill).


a_kind_of(canaries,birds).

a_kind_of(pet_canaries,canaries).

a_kind_of(pet_canaries,pets).

a_kind_of(pet_dogs,dogs).

a_kind_of(pet_dogs,pets).

a_kind_of(rottweilers,pet_dogs).


is_a(tweety,pet_canaries).

is_a(spike,rottweilers).

is_a(fido,pet_dogs).

is_a(john,person).

is_a(bill,person).
```

If these are loaded into Prolog, together with the inference rules, it will be seen that multiple inheritance works. We have:

```
?- value(fido,sound,S). S = bark ?
```

showing that fido inherits the sound bark from dogs,

```
?- value(fido,cute,V). V = true?
```

showing that fido inherits the cute is true from pets,

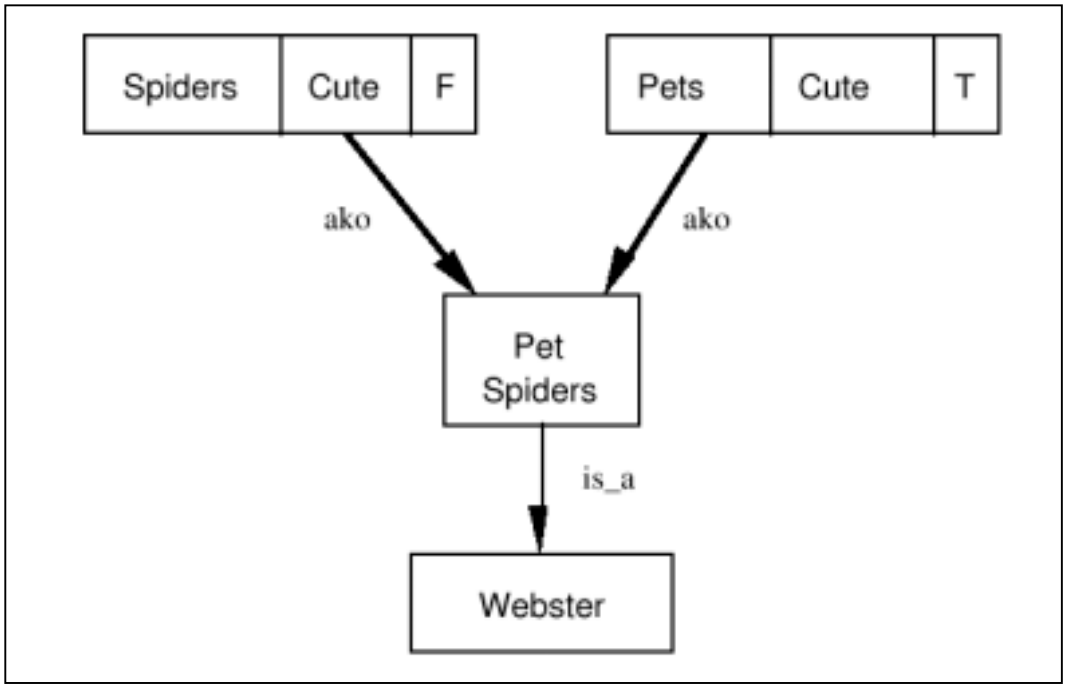
```
?- value(spike,cute,V). V = false?
```

showing that the cute is true property of pet_dogs is overridden in the rottweiler spike. Note that overrides may themselves be overridden. For example, in a classification of animals, molluscs typically have the property that they have shells. Cephalopods (octopuses and squids) are a subclass of mollusc which typically do not have shells, so the property has_shell=true is overridden. Nautiluses, however are a subclass of

cephalopods which typically do have shells, so the property is again overridden. This can easily be represented, in Prolog facts: attribute(molluscs,has_shell,true).

```
attribute(cephalopods,has_shell,false). attribute(nautiluses,has_shell,true).
a_kind_of(cephalopods,molluscs). a_kind_of(nautiluses,cephalopods).
```

A more tricky situation happens when with multiple inheritance an instance or a subclass inherits one property from one superclass and a contradictory property from another. This is often referred to as the "Nixon diamond" property, as it is frequently illustrated by the case of Richard Nixon being both a Quaker (a group whose members typically hold pacifist views) and a Republican (a group whose members typically do not hold pacifist views). As a similar example building from our previous examples, let us consider the case of pet spiders. As before we assume that pets are typically cute, but we will also assume that spiders are not typically cute. So are pet spiders typically cute or not?



In our Prolog representation, the answer will depend on the ordering of the clauses. If we have the ordering

```
ako(pet_spiders,spiders).

ako(pet_spiders,pets).
```

then using the rules defined above, we would get:

```
?- value(webster,cute,V). V = false ?
```

whereas if the order were

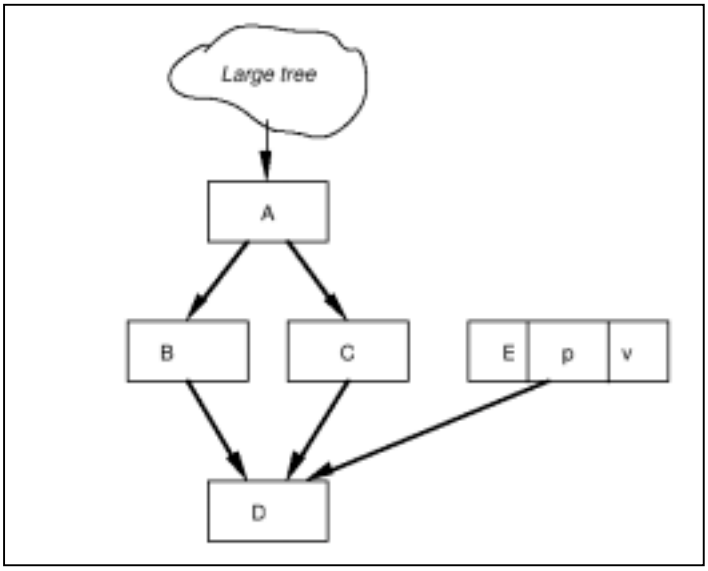
```
ako(pet_spiders,pets).

ako(pet_spiders,spiders).
```

we would get:

```
?- value(webster,cute,V). V = true ?
```

The reason for this is that the search for the cute attribute is a search through a tree with multiple inheritance, and our search rules if run under standard Prolog will use Prolog's depth-first left-to- right search of the tree. So if we list the fact that pet_spiders are a kind of spider before the fact that they are a kind of pet, the spider superclass will be searched for some attribute first, and vice versa. This is obviously a naïve way of solving the problem, more detailed discussion could be given about it, but at this stage it is sufficient to know of the problem. One difficulty, for example occurs if we want pet_spiders to inherit some conflicting attributes from pets and others from spiders. The way to resolve this is to specify default values for those attributes at the pet_spiders level.

Note that inheritance hierarchies with multiple inheritance can form graphs, since it is possible for something to be a subclass of two separate classes which are themselves subclasses of a single class. Consider for example:



In this case, class D multiply inherits from B, C and E with B and C having common superclass A. A further inherits from some large tree of superclasses. Suppose that property p is only found in class E. It will not be found in the search of the large tree. If we are searching for the p value of D, our naïve search would unnecessarily search the large tree twice, not find any reference to property p and only then look at E. In practice then, we would need to consider some of the graph search methods we considered earlier. We might also consider, for example, whether say a breadth-first search of the graph would be more appropriate than Prolog's built-in depth-first search.

Scripts

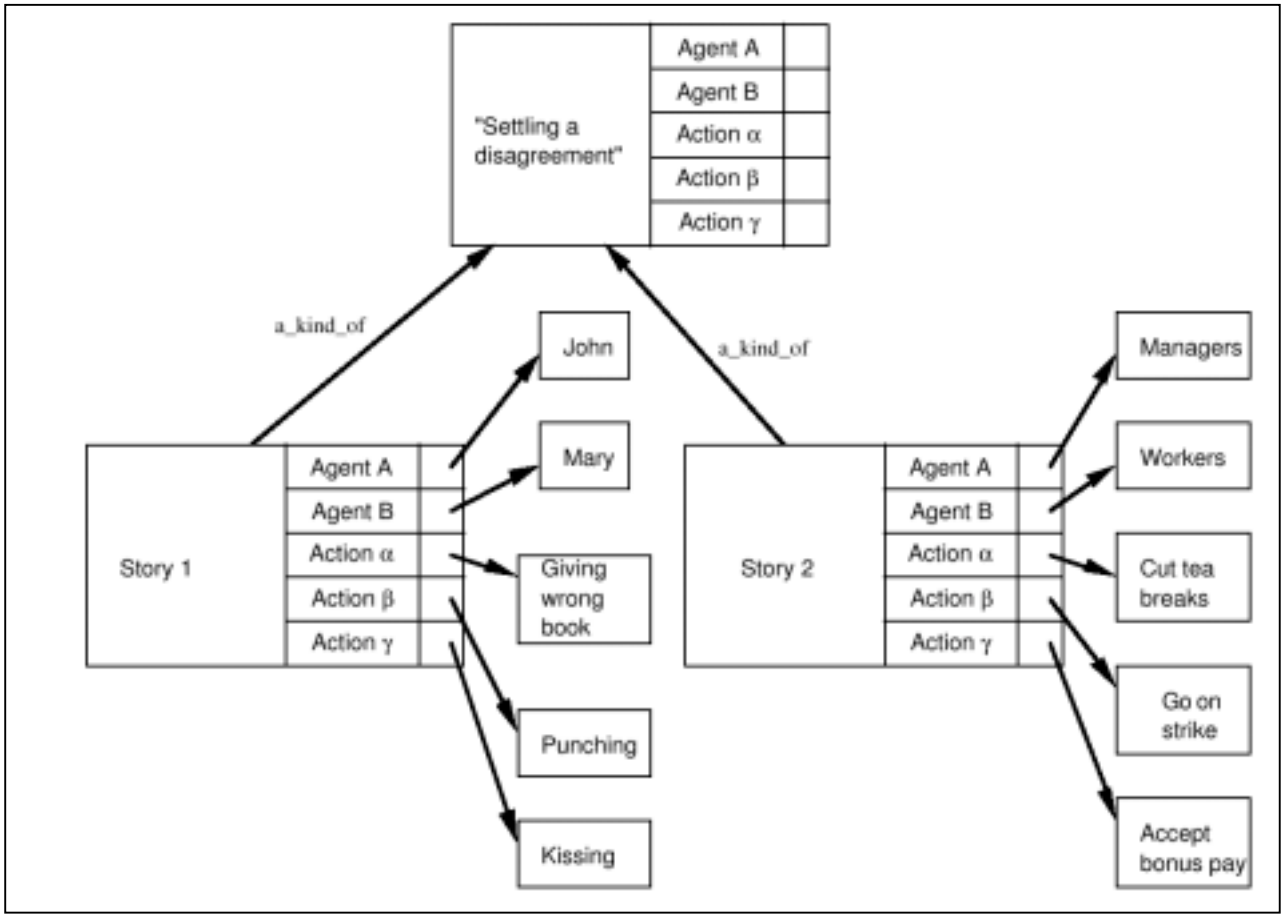
Scripts are a development of the idea of representing actions and events using semantic networks which we described above. With scripts the idea is that whole set of actions fall into stereotypical patterns. Scripts make use of the idea of defaults, with a class defining the roles in some action, and individual instances of the class having the roles filled in. This has been suggested as a way of analysing complete stories. For example, previously we had the story

"John gave Mary a book. The book was not the one Mary likes, so she punched John. That made her feel sorry for him, so she then kissed him".

This may be considered an instance of the script

> "A did action • for B. B didn't like •, so he/she/it/they did action • to hurt A. B then came to an agreement with A and did action • to make up".

In our previous example, A was John, B was Mary, □ was giving the wrong book, □ was punching, and □ was kissing. In another instance of the same script, A could be the factory managers, B the factory workers, □ could be cutting tea-break time, □ could be going on strike, and □ could be agreeing to accept a bonus payment.



The idea is that information on general points will be stored at the class level, which will enable us to answer questions on a variety of stories by relating them to a common theme.

Further Reading

A good coverage of the issues in this section is contained in:

H.Reichgelt *Knowledge Representation: An AI Perspective* Ablex Publishing Corporation 1991. A collection of reprints of original papers on the subject is:

R.J.Brachman and H.J.Levesque *Readings in Knowledge Representation* Morgan Kaufmann 1985. The subject in the context of object-oriented programming in:

G.Masini et al *Object Oriented Programming Languages* Academic Press 1991.

Further reading following from the section "The Case for Case" may be found in books on natural language processing, particularly those books with a good coverage of the semantic issues (many books on natural language processing are more concerned with the syntax i.e. saying whether a given sentence is grammatically correct or not, rather than the semantics i.e. determining the meaning of the sentence). Two books with a good coverage of semantics are:

M.D.Harris *Natural Language Processing* Prentice-Hall 1985. J.Allen *Natural Language Understanding* Benjamin/Cummings 1987.