2 Prolog: Representation

Chapter Objectives

Prolog's fundamental representations are described and built:
- Facts
- Rules
- The and, or, not, and imply connectives

The environment for Prolog is presented:
- The program as a data base of facts and relations between facts
- Predicates are for creating and modifying this data base

Prolog's procedural semantics is described with examples
- Pattern-matching

Left-to-right depth-first search
Backtracking on variable bindings
The built-in predicates for monitoring Prolog's execution are presented
- spy and trace

The list representation and recursive search are introduced
- Examples of member check and writing out lists

Representations for structured hierarchies are created in Prolog
- Semantic net and frame systems
- Inherited properties determined through recursive (tree) search

Chapter Contents

2.1 Introduction: Logic-Based Representation

Prolog and Logic

Prolog is a computer language that uses many of the representational strengths of the First-Order Predicate Calculus (Luger 2009, Chapter 2). Because Prolog has this representational power it can express general relationships between entities. This allows expressions such as "all females are intelligent" rather than the limited representations of the propositional calculus: "Kate is intelligent", "Sarah is intelligent", "Karen is intelligent", and so on for a very long time!

As in the Predicate Calculus, predicates offer the primary (and only) representational structure in Prolog. Predicates can have zero or more arguments, where their *arity* is the number of arguments. Functions may only be represented as the argument of a predicate; they cannot be a program statement in themselves. Prolog predicates have the usual `and`, `or`, `not` and `implies` connectives. The predicate representation along with its connectives is presented in Section 2.2.

Prolog also takes on many of the declarative aspects of the Predicate Calculus in the sense that a program is simply the set of all true predicates that describe a domain. The Prolog interpreter can be seen as a "theorem prover" that takes the user's query and determines whether or not it is true, as well as what variable substitutions might be required to make the query true. If the query is not true in the context of the program's specifications, the interpreter says "no."

2.2 Prolog Syntax

Facts, Rules and Connectives

Although there are numerous dialects of Prolog, the syntax used throughout this text is that of the original Warren and Pereira C-Prolog as described by Clocksin and Mellish (2003). We begin with the set of connectives that can take atomic predicates and join them with other expressions to make more complex relationships. There are, because of the usual keyboard conventions, a number of differences between Prolog and predicate calculus syntax. In C-Prolog, for example, the symbol :- replaces the ← of first-order predicate calculus. The Prolog connectives include:

ENGLISH	PREDICATE CALCULUS	Prolog
and	^	,
or	v	;
only if	←	:-
not	~	not

In Prolog, predicate names and bound variables are expressed as a sequence of alphanumeric characters beginning with an alphabetic. Variables are represented as a string of alphanumeric characters beginning (the first character, at least) with an uppercase alphabetic. Thus:

```
likes(X, susie).
```

or, better,

```
likes(Everyone, susie).
```

could represent the fact that "everyone likes Susie." Note that the scope of all variables is universal to that predicate, i.e., when a variable is used in a predicate it is understood that it is true for all the domain elements within its scope. For example,

```
likes(george, Y), likes(susie, Y).
```

represents the set of things (or people) liked by BOTH George and Susie.

Similarly, suppose it was desired to represent in Prolog the following relationships: "George likes Kate and George likes Susie." This could be stated as:

```
likes(george, kate), likes(george, susie).
```

Likewise, "George likes Kate or George likes Susie":

```
likes(george, kate); likes(george, susie).
```

Finally, "George likes Susie if George does not like Kate":

```
likes(george, susie) :- not(likes(george, kate)).
```

These examples show how the predicate calculus connectives are expressed in Prolog. The predicate names (likes), the number or order of parameters, and even whether a given predicate always has the same number of parameters are determined by the design requirements (the implicit "semantics") of the problem.

The form Prolog expressions take, as in the examples above, is a restricted form of the full predicate calculus called the "Horn Clause calculus." There are many reasons supporting this restricted form, most important is the power and computational efficiency of a *resolution refutation system*. For details see Luger (2009, Chapter 14).

A Simple Prolog Program

A Prolog program is a set of specifications in the first-order predicate calculus describing the objects and relations in a problem domain. The set of specifications is referred to as the *database* for that problem. The Prolog interpreter responds to questions about this set of specifications. Queries to the database are patterns in the same logical syntax as the database entries. The Prolog interpreter uses pattern-directed search to find whether these queries logically follow from the contents of the database.

The interpreter processes queries, searching the database in left to right depth-first order to find out whether the query is a logical consequence of the database of specifications. Prolog is primarily an interpreted language. Some versions of Prolog run in interpretive mode only, while others allow compilation of part or all of the set of specifications for faster execution. Prolog is an interactive language; the user enters queries in response to the Prolog prompt, "?-".

Let us describe a "world" consisting of George's, Kate's, and Susie's likes and dislikes. The database might contain the following set of predicates:

```
likes(george, kate).
likes(george, susie).
likes(george, wine).
likes(susie, wine).
likes(kate, gin).
likes(kate, susie).
```

This set of specifications has the obvious interpretation, or mapping, into the world of George and his friends. That world is a *model* for the database (Luger 2009, Section 2.3). The interpreter may then be asked questions:

```
?- likes(george, kate).
Yes
?- likes(kate, susie).
Yes
?- likes(george, X).
X = kate
;
X = Susie
;
X = wine
```

```
;
no
?- likes(george, beer).
no
```

Note first that in the request `likes(george, X)`, successive user prompts (`;`) cause the interpreter to return all the terms in the database specification that may be substituted for the `X` in the query. They are returned in the order in which they are found in the database: `kate` before `susie` before `wine`. Although it goes against the philosophy of nonprocedural specifications, a determined order of evaluation is a property of most interpreters implemented on sequential machines.

To summarize: further responses to queries are produced when the user prompts with the `;` (or). This forces the rejection of the current solution and a backtrack on the set of Prolog specifications for answers. Continued prompts force Prolog to find all possible solutions to the query. When no further solutions exist, the interpreter responds `no`.

This example also illustrates the *closed world assumption* or *negation as failure*. Prolog assumes that "anything is false whose opposite is not provably true." For the query `likes(george, beer)`, the interpreter looks for the predicate `likes(george, beer)` or some rule that could establish `likes(george, beer)`. Failing this, the request is false. Prolog assumes that all knowledge of the world is present in the database.

The closed world assumption introduces a number of practical and philosophical difficulties in the language. For example, failure to include a fact in the database often means that its truth is unknown; the closed world assumption treats it as false. If a predicate were omitted or there were a misspelling, such as `likes(george, beeer)`, the response remains `no`. Negation-as-failure issue is an important topic in AI research. Though negation-as-failure is a simple way to deal with the problem of unspecified knowledge, more sophisticated approaches, such as multi-valued logics (`true`, `false`, `unknown`) and nonmonotonic reasoning (see Luger 2009, Section 9.1), provide a richer interpretive context.

The Prolog expressions just seen are examples of *fact* specifications. Prolog also supports *rule* predicates to describe relationships between facts. We use the logical implication `:-` . For rules, only one predicate is permitted on the left-hand side of the *if* symbol `:-`, and this predicate must be a *positive literal*, which means it cannot have `not` in front of it. All predicate calculus expressions that contain logical implication must be reduced to this form, referred to as *Horn clause logic*. In Horn clause form, the left-hand side (conclusion) of an implication must be a single positive literal. The *Horn clause calculus* is equivalent to the full first-order predicate calculus for proofs by refutation (Luger 2009, Chapter 14).

Suppose we add to the specifications of the previous database a rule for determining whether two people are friends. This may be defined:

```
friends(X, Y) :- likes(X, Z), likes(Y, Z).
```

This expression might be interpreted as "`X` and `Y` are friends if there exists a `Z` such that `X` likes `Z` and `Y` likes `Z`." Two issues are important here. First,

because neither the predicate calculus nor Prolog has global variables, the scopes (extent of definition) of `X`, `Y`, and `Z` are limited to the `friends` rule. Second, values bound to, or unified with, `X`, `Y`, and `Z` are consistent across the entire expression. The treatment of the `friends` rule by the Prolog interpreter is seen in the following example.

With the `friends` rule added to the set of specifications of the preceding example, we can query the interpreter:

```
?- friends(george, susie).
yes
```

To solve this query, Prolog searches the database using the *backtrack* algorithm. Briefly, backtrack examines each predicate specification in the order that it was placed in the Prolog. If the variable bindings of the specification satisfy the query it accepts them. If they don't, the interpreter goes on to the next specification. If the interpreter runs into a dead end, i.e., no variable substitution satisfies it, then it backs up looking for other variable bindings for the predicates it has already satisfied. For example, using the predicate specifications of our current example, the query `friends(george, susie)` is unified with the conclusion of the rule `friends(X, Y) :- likes(X, Z), likes(Y, Z)`, with `X` as `george` and `Y` as `susie`. The interpreter looks for a `Z` such that `likes(george, Z)` is true and uses the first fact, with `Z` as `kate`.

The interpreter then tries to determine whether `likes(susie, kate)` is true. When it is found to be false, using the closed world assumption, this value for `Z` (`kate`) is rejected. The interpreter backtracks to find a second value for `Z`. `likes(george, Z)` then matches the second fact, with `Z` bound to `susie`. The interpreter then tries to match `likes(susie, susie)`. When this also fails, the interpreter goes back to the database for yet another value for `Z`. This time `wine` is found in the third predicate, and the interpreter goes on to show that `likes(susie, wine)` is true. In this case `wine` is the binding that ties `george` and `susie`.

It is important to state the relationship between universal and existential quantification in the predicate calculus and the treatment of variables in a Prolog program. When a variable is placed in the specifications of a Prolog database, it is universally quantified. For example, `likes(susie, Y)` means, according to the semantics of the previous examples, "Susie likes everyone." In the course of interpreting a query, any term, or list, or predicate from the domain of `Y`, may be bound to `Y`. Similarly, in the rule `friends(X, Y) :- likes(X, Z), likes(Y, Z)`, any `X`, `Y`, and `Z` that meets the specifications of the expression are used.

To represent an existentially quantified variable in Prolog, we may take two approaches. First, if the existential value of a variable is known, that value may be entered directly into the database. Thus, `likes(george, wine)` is an instance of `likes(george, Z)`.

Second, to find an instance of a variable that makes an expression true, we query the interpreter. For example, to find whether a `Z` exists such that `likes(george, Z)` is true, we put this query to the interpreter. It will

find whether a value of `Z` exists under which the expression is true. Some Prolog interpreters find all existentially quantified values; C-Prolog requires repeated user prompts (`;`), as shown previously, to get all values.

2.3 Creating, Changing, and Tracing a Prolog Computation

In building a Prolog program the database of specifications is created first. In an interactive environment the predicate `assert` can be used to add new predicates to the set of specifications. Thus:

```
?- assert(likes(david, sarah)).
```

adds this predicate to the computing specifications. Now, with the query:

```
?- likes(david, X).
X = sarah.
```

is returned. `assert` allows further control in adding new specifications to the database: `asserta(P)` asserts the predicate `P` at the beginning of all the predicates `P`, and `assertz(P)` adds `P` at the end of all the predicates named `P`. This is important for search priorities and building heuristics. To remove a predicate `P` from the database `retract(P)` is used. (It should be noted that in many Prologs `assert` can be unpredictable in that the exact entry time of the new predicate into the environment can vary depending on what other things are going on, affecting both the indexing of asserted clauses as well as backtracking.)

It soon becomes tedious to create a set of specifications using the predicates `assert` and `retract`. Instead, the good programmer takes her favorite editor and creates a file containing all the Prolog program's specifications. Once this file is created, call it `myfile`, and Prolog is called, then the file is placed in the database by the Prolog command `consult`. Thus:

```
?- consult(myfile).
yes
```

integrates the predicates in `myfile` into the database. A short form of the `consult` predicate, and better for adding multiple files to the database, uses the list notation, to be seen shortly:

```
?- [myfile].
yes
```

If there are any syntax errors in your Prolog code the `consult` operator will describe them at the time it is called.

The predicates `read` and `write` are important for user/system communication. `read(X)` takes the next term from the current input stream and binds it to `X`. Input expressions are terminated with a "." `write(X)` puts `X` in the output stream. If `X` is unbound then an integer preceded by an underline is printed (`_69`). This integer represents the internal bookkeeping on variables necessary in a theorem-proving environment (see Luger 2009, Chapter 14).

The Prolog predicates `see` and `tell` are used to read information from and place information into files. `see(X)` opens the file `X` and defines the current input stream as originating in `X`. If `X` is not bound to an available

file `see(X)` fails. Similarly, `tell(X)` opens a file for the output stream. If no file `X` exists, `tell(X)` creates a file named by the bound value of `X`. `seen(X)` and `told(X)` close the respective files.

A number of Prolog predicates are important in helping keep track of the state of the Prolog database as well as the state of computing about the database; the most important of these are `listing`, `trace`, and `spy`. If we use `listing(predicate_name)` where `predicate_name` is the name of a predicate, such as `friends` (above), all the clauses with that predicate name in the database are returned by the interpreter. Note that the number of arguments of the predicate is not indicated; in fact, all uses of the predicate, regardless of the number of arguments, are returned.

`trace` allows the user to monitor the progress of the Prolog interpreter. This monitoring is accomplished by printing to the output file every goal that Prolog attempts, which is often more information than the user wants to have. The tracing facilities in Prolog are often rather cryptic and take some study and experience to understand. The information available in a trace of a Prolog program usually includes the following:

The depth level of recursive calls (marked left to right on line).

When a goal is tried for the first time (sometimes `call` is used).

When a goal is successfully satisfied (with an `exit`).

When a goal has further matches possible (a `retry`).

When a goal fails because all attempts to satisfy it have failed

The goal `notrace` stops the exhaustive tracing.

When a more selective trace is required the goal `spy` is useful. This predicate takes a predicate name as argument but sometimes is defined as a prefix operator where the predicate to be monitored is listed after the operator. Thus, `spy member` causes the interpreter to print to output all uses of the predicate `member`. `spy` can also take a list of predicates followed by their arities: `spy[member/2, append/3]` monitors `member` with two arguments and `append` with three. `nospy` removes these spy points.

2.4 Lists and Recursion in Prolog

The previous subsections presented Prolog syntax with several simple examples. These examples introduced Prolog as an engine for computing with predicate calculus expressions (in Horn clause form). This is consistent with all the principles of predicate calculus inference presented in Luger (2009, Chapter 2). Prolog uses unification for pattern matching and returns the bindings that make an expression true. These values are unified with the variables in a particular expression and are not bound in the global environment.

Recursion is the primary control mechanism for Prolog programming. We will demonstrate this with several examples. But first we consider some simple list-processing examples. The list is a data structure consisting of ordered sets of elements (or, indeed, lists). Recursion is the natural way to process the list structure. Unification and recursion come together in list

processing in Prolog. The set of elements of a list are enclosed by brackets, [ ], and are separated by commas. Examples of Prolog lists are:

```
[1, 2, 3, 4]
[[george, kate], [allen, amy], [richard, shirley]]
[tom, dick, harry, fred]
[ ]
```

The first elements of a list may be separated from the tail of the list by the bar operator, |. The tail of a list is the list with its first element removed. For instance, when the list is `[tom,dick,harry,fred]`, the first element is `tom` and the tail is the list `[dick, harry, fred]`. Using the vertical bar operator and unification, we can break a list into its components:

```
If [tom, dick, harry, fred] is matched to [X | Y],
then X = tom and Y = [dick, harry, fred].
If [tom,dick,harry,fred] is matched to the pattern
[X, Y | Z], then X = tom , Y = dick , and Z =
[harry, fred].
If [tom, dick, harry, fred] is matched to [X, Y, Z |
W], then X = tom, Y = dick, Z = harry, and W =
[fred].
If [tom, dick, harry, fred] is matched to [W, X, Y,
Z | V], then W = tom, X = dick, Y = harry, Z = fred,
and V = [ ].
[tom, dick, harry, fred] will not match [V, W, X, Y,
Z | U].
[tom, dick, harry, fred] will match [tom, X |
[harry, fred]], to give X = dick.
```

Besides "tearing lists apart" to get at particular elements, unification can be used to "build" the list structure. For example, if `X = tom`, `Y = [dick]` when `L` unifies with `[X | Y]`, then `L` will be bound to `[tom, dick]`. Thus terms separated by commas before the | are all elements of the list, and the structure after the | is always a list, the tail of the list.

Let's take a simple example of recursive processing of lists: the `member` check. We define a predicate to determine whether an item, represented by `X`, is in a list. This predicate `member` takes two arguments, an element and a list, and is true if the element is a member of the list. For example:

```
?- member(a, [a, b, c, d, e]).
yes
?- member(a, [1, 2, 3, 4]).
no
?- member(X, [a, b, c]).
X = a
;
X = b
;
X = c
```

```
;
no
```

To define member recursively, we first test if **X** is the first item in the list:

```
member(X, [X | T]).
```

This tests whether **X** and the first element of the list are identical. Not that this pattern will match no matter what X is bound to: an atom, a list, whatever! If the two are not identical, then it is natural to check whether **X** is an element of the rest (**T**) of the list. This is defined by:

```
member(X, [Y | T]) :- member(X, T).
```

The two lines of Prolog for checking list membership are then:

```
member(X, [X | T]).
member(X, [Y | T]) :- member(X, T).
```

This example illustrates the importance of Prolog's built-in order of search with the terminating condition placed before the recursive call, that is, to be tested before the algorithm recurs. If the order of the predicates is reversed, the terminating condition may never be checked. We now trace `member(c,[a,b,c])`, with numbering:

```
1: member(X, [X | T]).
2: member(X, [Y | T]) :- member(X, T).
?- member(c, [a, b, c]).
    call 1. fail, since c <> a
    call 2. X = c, Y = a, T = [b, c],
                  member(c, | [b,c])?
         call 1. fail, since c <> b
         call 2. X = c, Y = b, T = [c],
                       member(c, | [c])?
              call 1. success, c = c
         yes (to second call 2.)
    yes (to first call 2.)
yes
```

Good Prolog style suggests the use of *anonymous variables*. These serve as an indication to the programmer and interpreter that certain variables are used solely for pattern-matching purposes, with the variable binding itself not part of the computation process. Thus, when we test whether the element **X** is the same as the first item in the list we usually say: `member(X, [X|_])`. The use of the _ indicates that even though the tail of the list plays a crucial part in the unification of the query, the content of the tail of the list is unimportant. In the `member` check the anonymous variable should be used in the recursive statement as well, where the value of the head of the list is unimportant:

```
member(X, [X | _]).
member(X, [_ | T]) :- member(X, T).
```

Writing out a list one element to a line is a nice exercise for understanding both lists and recursive control. Suppose we wish to write out the list `[a,b,c,d]`. We could define the recursive command:

```
writelist([ ]).
writelist([H | T]) :- write(H), nl, writelist(T).
```

This predicate writes one element of the list on each line, as `nl` requires the output stream controller to begin a new line.

If we wish to write out a list in reversed order the recursive predicate must come before the `write` command. This guarantees that the list is traversed to the end before any element is written. At that time the last element of the list is written followed by each preceding element as the recursive control comes back up to the top. A reverse write of a list would be:

```
reverse_writelist([ ]).
reverse_writelist([H | T]) :- reverse_writelist(T),
     write(H), nl.
```

The reader should run `writelist` and `reverse_writelist` with trace to observe the behavior of these predicates.

2.5 Structured Representations and Inheritance Search

Semantic Nets in Prolog

Structured representations are an important component of the AI representational toolkit (Collins and Quillian 1969, Luger 2009). They also support many of the design patterns mentioned in Chapter 1. In this and the following section we consider two structured representations, the *semantic net*, and *frames* that are used almost ubiquitously in AI. We now propose a simple semantic network representational structure in Prolog and use recursive search to implement inheritance. Our language ignores the important distinction between classes and instances. This restriction simplifies the implementation of inheritance.

In the semantic net of Figure 2.1, nodes represent individuals such as the canary `tweety` and classes such as `ostrich`, `crow`, `robin`, `bird`, and `vertebrate`. `isa` links represent the class hierarchy relationship. We adopt canonical forms for the data relationships within the net. We use an `isa(Type, Parent)` predicate to indicate that `Type` is a member of `Parent` and a `hasprop(Object, Property, Value)` predicate to represent property relations. `hasprop` indicates that `Object` has `Property` with `Value`. `Object` and `Value` are nodes in the network, and `Property` is the name of the link that joins them.

A partial list of predicates describing the bird hierarchy of Figure 2.1 is:

```
isa(canary, bird).    hasprop(tweety, color, white)
isa(robin, bird).     hasprop(robin, color, red).
isa(ostrich, bird).   hasprop(canary, color, yellow).
isa(penguin, bird).   hasprop(penguin, color, brown).
isa(bird, animal).    hasprop(bird, travel, fly).
isa(fish, animal).    hasprop(ostrich, travel, walk).
isa(opus, penguin).   hasprop(fish, travel, swim).
isa(tweety, canary). hasprop(robin, sound, sing).
hasprop(canary, sound, sing).
hasprop(bird, cover, feathers).
hasprop(animal, cover, skin).
```

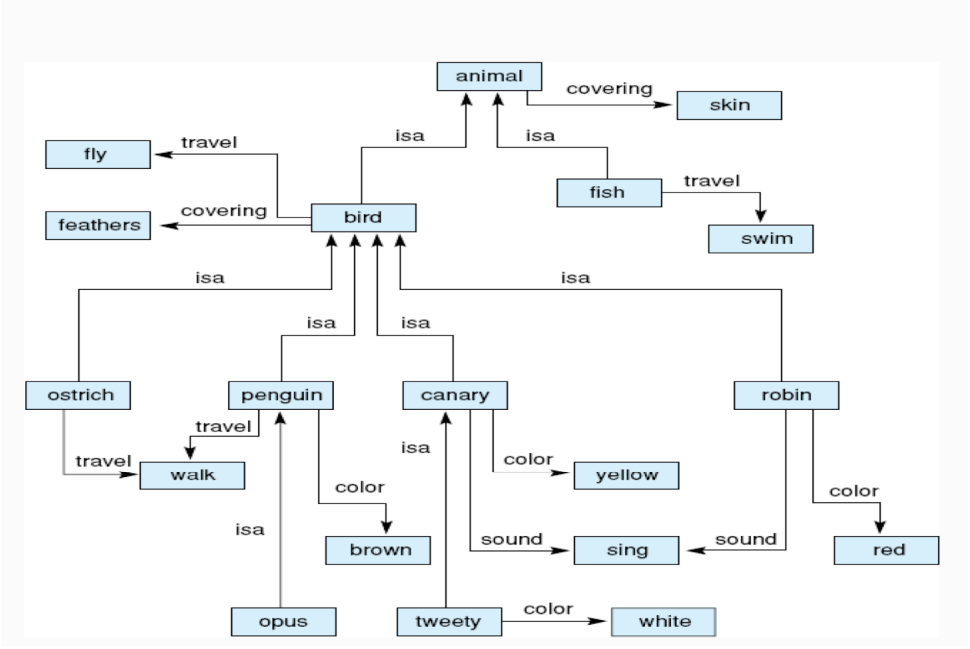

Figure 2.1 A semantic net for a bird hierarchy reflecting the Prolog code.

We create a recursive search algorithm to find whether an object in our semantic net has a particular property. Properties are stored in the net at the most general level at which they are true. Through inheritance, an individual or subclass acquires the properties of its superclasses. Thus the property `fly` holds for `bird` and all its subclasses. Exceptions are located at the specific level of the exception. Thus, `ostrich` and `penguin` travel by walking instead of flying. The `hasproperty` predicate begins search at a particular object. If the information is not directly attached to that object, `hasproperty` follows `isa` links to superclasses. If no more superclasses exist and `hasproperty` has not located the property, it fails.

```
haspropery(Object, Property, Value) :-
     hasprop(Object, Property, Value).
haspropery(Object, Property, Value) :-
     isa(Object, Parent),
     hasproperty(Parent, Property, Value).
```

`hasproperty` searches the inheritance hierarchy in a depth-first fashion. In the next section, we show how inheritance can be applied to a frame-based representation with both single and multiple-inheritance relations.

Frames in Prolog

Semantic nets can be partitioned, with additional information added to node descriptions, to give them a frame-like structure (Minsky 1975, Luger 2009). We present the bird example again using frames, where each frame represents a collection of relationships of the semantic net and the `isa` slots of the frame define the frame hierarchy as in Figure 2.2.

The first slot of each of the frames name that node, for example, `name(tweety)` or `name(vertebrate)`. The second slot gives the inheritance links between that node and its parents. Because our example

has a tree structure, each node has only one link, the `isa` predicate with one argument. The third slot in the node's frame is a list of features that describe that node. In this list we use any Prolog predicate such as `flies`, `feathers`, or `color(brown)`. The final slot in the frame is the list of exceptions and default values for the node, again either a single word or predicate indicating a property.

In our frame language, each frame organizes its slot names into lists of properties and default values. This allows us to distinguish these different types of knowledge and give them different behaviors in the inheritance hierarchy. Although our implementation allows subclasses to inherit properties from both lists, other representations are possible and may be useful in certain applications. We may wish to specify that only default values are inherited. Or we may wish to build a third list containing the properties of the class itself rather than the members, sometimes called *class values*. For example, we may wish to state that the class `canary` names a species of `songbird`. This should not be inherited by subclasses or instances: `tweety` does not name a species of `songbird`. Further extensions to this example are suggested in the exercises.

We now represent the relationships in Figure 2.2 with the Prolog fact predicate `frame` with four arguments. We may use the methods suggested in Chapter 5 to check the parameters of the frame predicate for appropriate type, for instance, to ensure that the third frame slot is a list that contains only values from a fixed list of properties.

```
frame(name(bird),
     isa(animal),
     [travel(flies), feathers],
     [ ]).
frame(name(penguin),
     isa(bird),
     [color(brown)],
     [travel(walks)]).
frame(name(canary),
     isa(bird),
     [color(yellow), call(sing)],
     [size(small)]).
frame(name(tweety),
     isa(canary),
     [ ],
     [color(white)]).
```

Once the full set of descriptions and inheritance relationships are defined for the frame of Figure 2.2, we create procedures to infer properties from this representation:

```
get(Prop, Object) :-
     frame(name(Object), _, List_of_properties,_),
     member(Prop, List_of_properties).
```

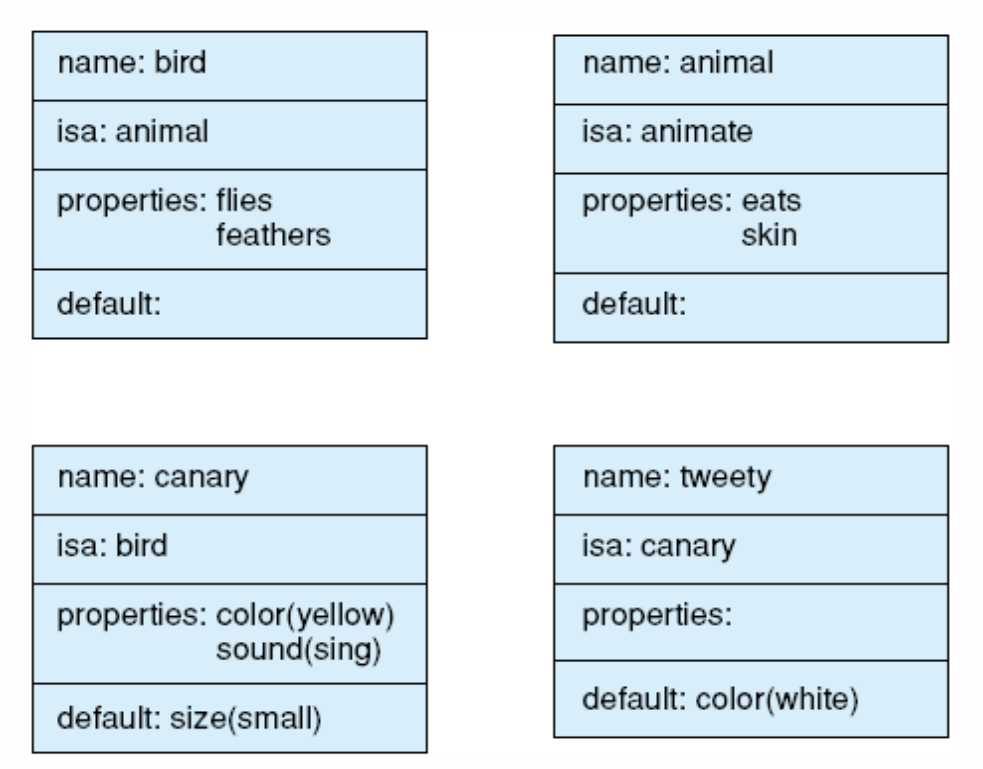

Figure 2.2 A frame system reflecting the Prolog code in the text.

```
get(Prop, Object) :-
     frame(name(Object), _, _ List_of_defaults),
     member(Prop, List_of_defaults).
get(Prop, Object) :-
     frame(name(Object), isa(Parent), _, _),
     get(Prop, Parent).
```

If the frame structure allows multiple inheritance of properties, we make this change both in our representation and in our search strategy. First, in the frame representation we make the argument of the `isa` predicate a list of superclasses of the `Object`. Thus, each superclass in the list is a parent of the entity named in the first argument of `frame`. If `opus` is a `penguin` and a `cartoon_char` we represent this:

```
frame(name(opus),
     isa([penguin, cartoon_char]),
     [color(black)],
     [ ]).
```

Now, we test for properties of `opus` by recurring up the `isa` hierarchy for both `penguin` and `cartoon_char`. We add the following `get` definition between the third and fourth `get` predicates of the previous example.

```
get(Prop, Object) :-
     frame(name(Object), isa(List), _, _),
     get_multiple(Prop, List).
```

We define get_multiple by:

```
get_multiple(Prop, [Parent  _]) :-
     get(Prop, Parent).
get_multiple(Prop, [_ Rest]) :-
     get_multiple(Prop, Rest).
```

With this inheritance preference, properties of `penguin` and its superclasses will be examined before those of `cartoon_char`.

Finally, any Prolog procedure may be attached to a frame slot. As we have built the frame representation in our examples, this would entail adding a Prolog rule, or list of Prolog rules, as a parameter of frame. This is accomplished by enclosing the entire rule in parentheses, as we will see for rules in `exshell` in Chapter 6, and making this structure an argument of the frame predicate. For example, we could design a list of response rules for `opus`, giving him different responses for different questions.

This list of rules, each rule in parentheses, would then become a parameter of the frame and, depending on the value of `X` passed to the `opus` frame, would define the appropriate response. More complex examples could be rules describing the control of a thermostat or creating a graphic image appropriate to a set of values. Examples of this are presented in both Lisp (Chapter 17) and Java (Chapter 21) where attached procedures, often called *methods*, play an important role in object-oriented representations.

Exercises

1. Create a relational database in Prolog. Represent the data tuples as facts and the constraints on the data tuples as rules. Suitable examples might be from stock in a department store or records in a personnel office.

2. Write the "member check" program in Prolog. What happens when an item is not in the list? Query to the "member" specification to break a list into its component elements.

3. Design a Prolog program `unique(Bag, Set)` that takes a *bag* (a list that may contain duplicate elements) and returns a *set* (no elements are repeated).

4. Write a Prolog program to count the elements in a list (a list within the list counts as one element). Write a program to count the atoms in a list (count the elements within any sublist). Hint: several meta-predicates such as `atom()` can be helpful.

5. Implement a frame system with inheritance that supports the definition of three kinds of slots: properties of a class that may be inherited by subclasses, properties that are inherited by instances of the class but not by subclasses, and properties of the class and its subclasses that are not inherited by instances (class properties). Discuss the benefits, uses, and problems with this distinction.