chapter 4

Using Structures: Example Programs

Data structures, with matching, backtracking and arithmetic, are a powerful programming tool. In this chapter we will develop the skill of using this tool through programming examples: retrieving structured information from a database, simulating a non-deterministic automaton, travel planning, and eight queens on the chessboard. We will also see how the principle of data abstraction can be carried out in Prolog. The programming examples in this chapter can be read selectively.

4.1 Retrieving structured information from a database

This exercise develops techniques of representing and manipulating structured data objects. It also illustrates Prolog as a natural database query language.

A database can be naturally represented in Prolog as a set of facts. For example, a database about families can be represented so that each family is described by one clause. Figure 4.1 shows how the information about each family can be structured. Each family has three components: husband, wife and children. As the number of children varies from family to family the children are represented by a list that is capable of accommodating any number of items. Each person is, in turn, represented by a structure of four components: name, surname, date of birth, job. The job information is 'unemployed', or it specifies the working organization and salary. The family of Figure 4.1 can be stored in the database by the clause:

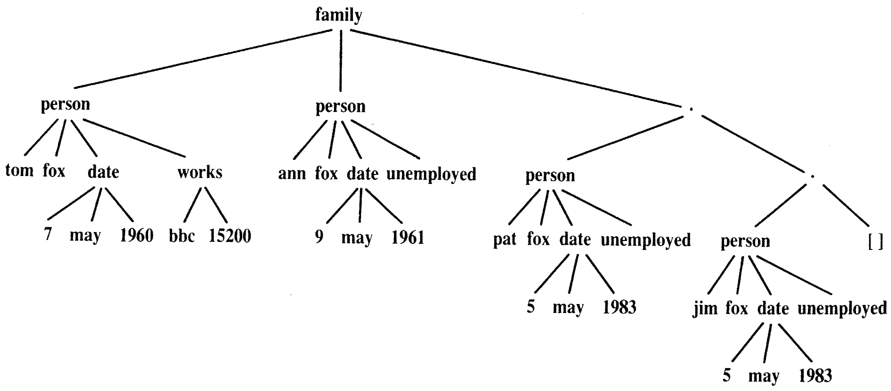

Figure 4.1 Structuring information about the family.

```
family(
   person( tom, fox, date(7,may,1960), works(bbc,15200) ),
   person( ann, fox, date(9,may,1961), unemployed),
   [ person( pat, fox, date(5,may,1983), unemployed),
     person( jim, fox, date(5,may,1983), unemployed) ] ).
```

Our database would then be comprised of a sequence of facts like this describing all families that are of interest to our program.

Prolog is, in fact, a very suitable language for retrieving the desired information from such a database. One nice thing about Prolog is that we can refer to objects without actually specifying all the components of these objects. We can merely indicate the *structure* of objects that we are interested in, and leave the particular components in the structures unspecified or only partially specified. Figure 4.2 shows some examples. So we can refer to all Armstrong families by:

```
family( person( _, armstrong, _, _), _, _)
```

The underscore characters denote different anonymous variables; we do not care about their values. Further, we can refer to all families with three children by the term:

```
family( _, _, [_, _, _] )
```

To find all married women that have at least three children we can pose the question:

```
?- family( _, person( Name, Surname, _, _), [_, _, _ | _] ).
```

The point of these examples is that we can specify objects of interest not by their content, but by their structure. We only indicate their structure and leave their arguments as unspecified slots.

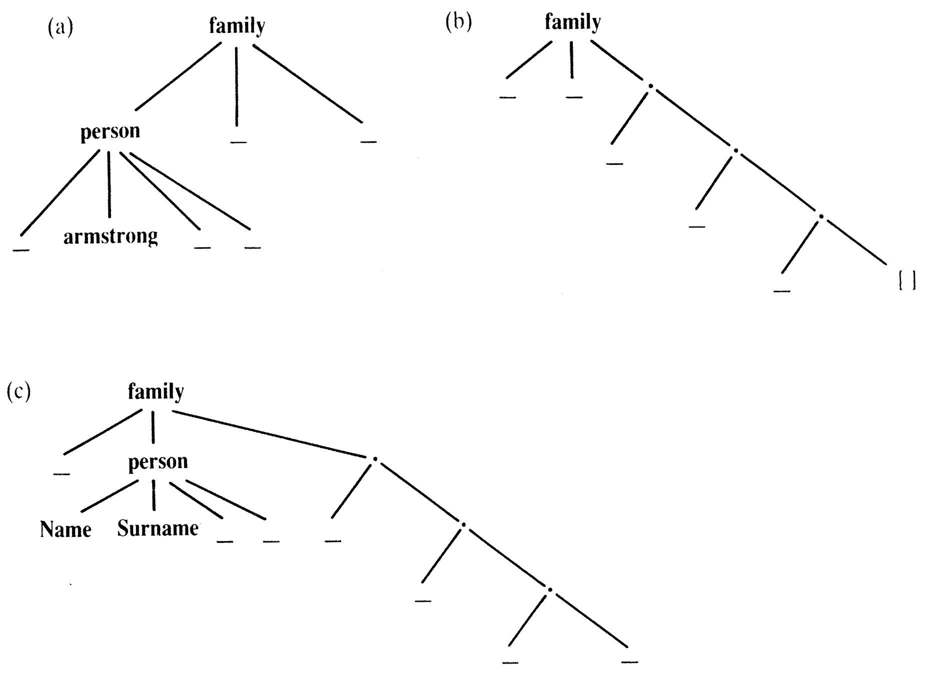

Figure 4.2 Specifying objects by their structural properties: (a) any Armstrong family; (b) any family with exactly three children; (c) any family with at least three children. Structure (c) makes provision for retrieving the wife's name through the instantiation of the variables **Name** and **Surname**.

We can provide a set of procedures that can serve as a utility to make the interaction with the database more comfortable. Such utility procedures could be part of the user interface. Some useful utility procedures for our database are:

```
husband( X) :-                      % X is a husband
  family( X, _, _).

wife( X) :-                         % X is a wife
  family( _, X, _).

child( X) :-                        % X is a child
  family( _, _, Children),
  member( X, Children).             % X in list Children

exists( Person) :-                  % Any person in the database
  husband( Person)
  ;
  wife( Person)
  ;
  child( Person).
```

```
dateofbirth( person( _, _, Date, _), Date).
salary( person( _, _, _, works( _, S) ), S).      % Salary of working person
salary( person( _, _, _, unemployed), 0).         % Salary of unemployed
```

We can use these utilities, for example, in the following queries to the database:

- Find the names of all the people in the database:

```
?- exists( person( Name, Surname, _, _) ).
```

- Find all children born in 2000:

```
?- child( X),
   dateofbirth( X, date( _, _, 2000) ).
```

- Find all employed wives:

```
?- wife( person( Name, Surname, _, works( _, _) ) ).
```

- Find the names of unemployed people who were born before 1973:

```
?- exists( person( Name, Surname, date( _, _, Year), unemployed) ),
   Year < 1973.
```

- Find people born before 1960 whose salary is less than 8000:

```
?- exists( Person),
   dateofbirth( Person, date( _, _, Year) ),
   Year < 1960,
   salary( Person, Salary),
   Salary < 8000.
```

- Find the names of families with at least three children:

```
?- family( person( _, Name, _, _), _, [_, _, _ | _] ).
```

To calculate the total income of a family it is useful to define the sum of salaries of a list of people as a two-argument relation:

```
total( List_of_people, Sum_of_their_salaries)
```

This relation can be programmed as:

```
total( [], 0).                          % Empty list of people

total( [Person | List], Sum) :-
   salary( Person, S),                  % S: salary of first person
   total( List, Rest),                  % Rest: sum of salaries of others
   Sum is S + Rest.
```

The total income of families can then be found by the question:

```
?- family( Husband, Wife, Children),
   total( [Husband, Wife | Children], Income).
```

Let the **length** relation count the number of elements of a list, as defined in Section 3.4. Then we can specify all families that have an income per family member of less than 2000 by:

```
?- family( Husband, Wife, Children),
   total( [Husband, Wife | Children], Income),
   length( [Husband, Wife | Children], N),        % N: size of family
   Income/N < 2000.
```

Exercises

4.1 Write queries to find the following from the family database:

(a) names of families without children;

(b) all employed children;

(c) names of families with employed wives and unemployed husbands;

(d) all the children whose parents differ in age by at least 15 years.

4.2 Define the relation

```
twins( Child1, Child2)
```

to find twins in the family database.

4.2 Doing data abstraction

Data abstraction can be viewed as a process of organizing various pieces of information into natural units (possibly hierarchically), thus structuring the information into some conceptually meaningful form. Each such unit of information should be easily accessible in the program. Ideally, all the details of implementing such a structure should be invisible to the user of the structure – the programmer can then just concentrate on objects and relations between them. The point of the process is to make the use of information possible without the programmer having to think about the details of how the information is actually represented.

Let us discuss one way of carrying out this principle in Prolog. Consider our family example of the previous section again. Each family is a collection of pieces of information. These pieces are all clustered into natural units such as a person or a family, so they can be treated as single objects. Assume again that the family information is structured as in Figure 4.1. In the previous section, each family was represented by a Prolog clause. Here, a family will be represented as a structured object, for example:

```
FoxFamily = family( person( tom, fox, _, _), _, _)
```

Let us now define some relations through which the user can access particular components of a family without knowing the details of Figure 4.1. Such relations can be called *selectors* as they select particular components. The name of such a selector relation will be the name of the component to be selected. The relation will have two arguments: first, the object that contains the component, and second, the component itself:

```
selector_relation( Object, Component_selected)
```

Here are some selectors for the family structure:

```
husband( family( Husband, _, _), Husband).

wife( family( _, Wife, _), Wife).

children( family( _, _, ChildList), ChildList).
```

We can also define selectors for particular children:

```
firstchild( Family, First) :-
  children( Family, [First | _] ).

secondchild( Family, Second) :-
  children( Family, [_, Second | _] ).

...
```

We can generalize this to selecting the Nth child:

```
nthchild( N, Family, Child) :-
  children( Family, ChildList),
  nth_member( N, ChildList, Child).        % Nth element of a list
```

Another interesting object is a person. Some related selectors according to Figure 4.1 are:

```
firstname( person( Name, _, _, _), Name).

surname( person( _, Surname, _, _), Surname).

born( person( _, _, Date, _), Date).
```

How can we benefit from selector relations? Having defined them, we can now forget about the particular way that structured information is represented. To create and manipulate this information, we just have to know the names of the selector relations and use these in the rest of the program. In the case of complicated representations, this is easier than always referring to the representation explicitly. In our family example in particular, the user does not have to know that the children are represented as a list. For example, assume that we want to say that Tom Fox and Jim Fox belong to the same family and that Jim is the second child of Tom. Using the selector relations above, we can define two persons, call them **Person1** and **Person2**, and the family. The following list of goals does this:

```
firstname( Person1, tom), surname( Person1, fox),      % Person1 is Tom Fox
firstname( Person2, jim), surname( Person2, fox),      % Person2 is Jim Fox
husband( Family, Person1),
secondchild( Family, Person2)
```

As a result, the variables **Person1**, **Person2** and **Family** are instantiated as:

```
Person1 = person( tom, fox, _, _)
Person2 = person( jim, fox, _, _)
Family = family( person( tom, fox, _, _), _, [ _, person( jim, fox) | _])
```

The use of selector relations also makes programs easier to modify. Imagine that we would like to improve the efficiency of a program by changing the representation of data. All we have to do is to change the definitions of the selector relations, and the rest of the program will work unchanged with the new representation.

Exercise

4.3 Complete the definition of **nthchild** by defining the relation

```
nth_member( N, List, X)
```

which is true if X is the Nth member of **List**.

4.3 Simulating a non-deterministic automaton

This exercise shows how an abstract mathematical construct can be translated into Prolog. In addition, our resulting program will turn out to be much more flexible than initially intended.

A *non-deterministic finite automaton* is an abstract machine that reads as input a string of symbols and decides whether to *accept* or to *reject* the input string. An automaton has a number of *states* and it is always in one of the states. It can change its state by moving from the current state to another state. The internal structure of the automaton can be represented by a transition graph such as that in Figure 4.3. In this example, s_1, s_2, s_3 and s_4 are the *states* of the automaton. Starting from the initial state (s_1 in our example), the automaton moves from state to state while reading the input string. Transitions depend on the current input symbol, as indicated by the arc labels in the transition graph.

A transition occurs each time an input symbol is read. Note that transitions can be non-deterministic. In Figure 4.3, if the automaton is in state s_1 and the current input symbol is a then it can transit into s_1 or s_2. Some arcs are labelled *null* denoting the 'null symbol'. These arcs correspond to 'silent moves' of the automaton. Such a

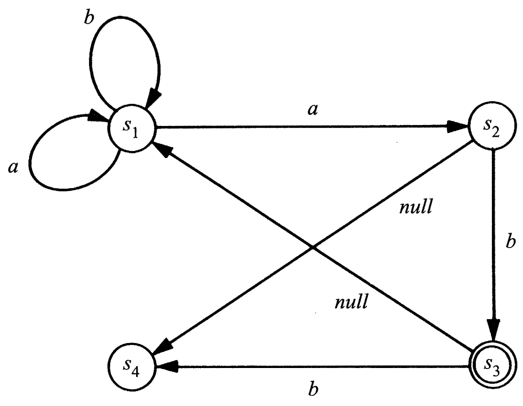

Figure 4.3 An example of a non-deterministic finite automaton.

move is said to be *silent* because it occurs without any reading of input, and the observer, viewing the automaton as a black box, will not be able to notice that any transition has occurred.

The state s_3 is double circled, which indicates that it is a *final state*. The automaton is said to *accept* the input string if there is a transition path in the graph such that

(1) it starts with the initial state,

(2) it ends with a final state, and

(3) the arc labels along the path correspond to the complete input string.

It is entirely up to the automaton to decide which of the possible moves to execute at any time. In particular, the automaton may choose to make or not to make a silent move, if it is available in the current state. But abstract non-deterministic machines of this kind have a magic property: if there is a choice then they always choose a 'right' move; that is, a move that leads to the acceptance of the input string, if such a move exists. The automaton in Figure 4.3 will, for example, accept the strings *ab* and *aabaab*, but it will reject the strings *abb* and *abba*. It is easy to see that this automaton accepts any string that terminates with *ab*, and rejects all others.

In Prolog, an automaton can be specified by three relations:

(1) a unary relation **final** which defines the final states of the automaton;

(2) a three-argument relation **trans** which defines the state transitions so that

trans(S1, X, S2)

means that a transition from a state S1 to S2 is possible when the current input symbol X is read;

(3) a binary relation

```
silent( S1, S2)
```

meaning that a silent move is possible from S1 to S2.

For the automaton in Figure 4.3 these three relations are:

```
final( s3).

trans( s1, a, s1).
trans( s1, a, s2).
trans( s1, b, s1).
trans( s2, b, s3).
trans( s3, b, s4).

silent( s2, s4).
silent( s3, s1).
```

We will represent input strings as Prolog lists. So the string *aab* will be represented by [a,a,b]. Given the description of the automaton, the simulator will process a given input string and decide whether the string is accepted or rejected. By definition, the non-deterministic automaton accepts a given string if (starting from an initial state), after having read the whole input string, the automaton can (possibly) be in its final state. The simulator is programmed as a binary relation, **accepts**, which defines the acceptance of a string from a given state. So

```
accepts( State, String)
```

is true if the automaton, starting from the state **State** as initial state, accepts the string **String**. The **accepts** relation can be defined by three clauses. They correspond to the following three cases:

(1) The empty string, [], is accepted from a state **State** if **State** is a final state.

(2) A non-empty string is accepted from **State** if reading the first symbol in the string can bring the automaton into some state **State1**, and the rest of the string is accepted from **State1**. Figure 4.4(a) illustrates.

(3) A string is accepted from **State** if the automaton can make a silent move from **State** to **State1** and then accept the (whole) input string from **State1**. Figure 4.4(b) illustrates.

These rules can be translated into Prolog as:

```
accepts( State, [] ) :-                 % Accept empty string
  final( State).

accepts( State, [X | Rest] ) :-         % Accept by reading first symbol
  trans( State, X, State1),
  accepts( State1, Rest).

accepts( State, String) :-              % Accept by making silent move
  silent( State, State1),
  accepts( State1, String).
```

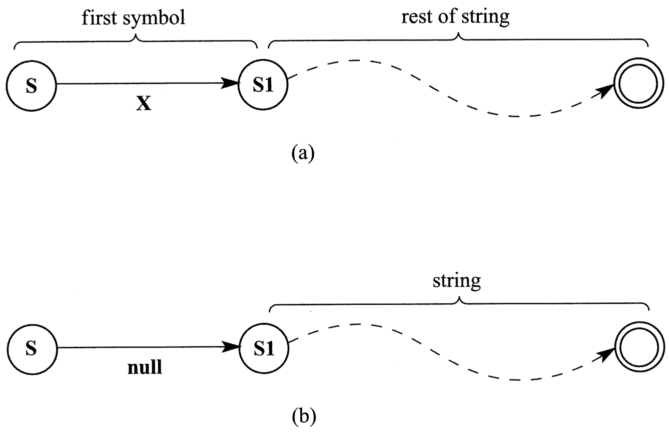

Figure 4.4 Accepting a string: (a) by reading its first symbol X; (b) by making a silent move.

The program can be asked, for example, about the acceptance of the string *aaab* by:

```
?- accepts( s1, [a,a,a,b] ).

yes
```

As we have already seen, Prolog programs are often able to solve more general problems than problems for which they were originally developed. In our case, we can also ask the simulator which state our automaton can be in initially so that it will accept the string *ab*:

```
?- accepts( S, [a,b] ).

S = s1;

S = s3
```

Amusingly, we can also ask: What are all the strings of length 3 that are accepted from state s_1?

```
?- accepts( s1, [X1,X2,X3] ).

X1 = a
X2 = a
X3 = b;

X1 = b
X2 = a
X3 = b;

no
```

If we prefer the acceptable input strings to be typed out as lists then we can formulate the question as:

```
?- String = [_, _, _], accepts( s1, String).
String = [a,a,b];
String = [b,a,b];
no
```

We can make further experiments asking even more general questions, such as: From what states will the automaton accept input strings of length 7?

Further experimentation could involve modifications in the structure of the automaton by changing the relations **final**, **trans** and **silent**. The automaton in Figure 4.3 does not contain any cyclic 'silent path' (a path that consists only of silent moves). If in Figure 4.3 a new transition

```
silent( s1, s3)
```

is added then a 'silent cycle' is created. But our simulator may now get into trouble. For example, the question

```
?- accepts( s1, [a] ).
```

would induce the simulator to cycle in state s_1 indefinitely, all the time hoping to find some way to the final state.

Exercises

4.4 Why could cycling not occur in the simulation of the original automaton in Figure 4.3, when there was no 'silent cycle' in the transition graph?

4.5 Cycling in the execution of **accepts** can be prevented, for example, by counting the number of moves made so far. The simulator would then be requested to search only for paths of some limited length. Modify the **accepts** relation this way. Hint: Add a third argument: the maximum number of moves allowed:

```
accepts( State, String, MaxMoves)
```

4.4 Travel agent

In this section we will construct a program that gives advice on planning air travel. The program will be a rather simple advisor, yet it will be able to answer some useful questions, such as:

- What days of the week is there a direct evening flight from Ljubljana to London?
- How can I get from Ljubljana to Edinburgh on Thursday?