Chapter 7

More Example Programs

Each section of this chapter deals with a particular application of Prolog programming. We suggest that you read all of the sections in this chapter. Do not be concerned if you do not understand the purpose of a program because you are not acquainted with the particular application. For example, only those readers who have been introduced to Calculus will appreciate the value of symbolic differentiation. Read it anyway, because the program for finding symbolic derivatives demonstrates how to use pattern matching to transform one kind of structure (an arithmetic expression) into another one. What is important is to gain an understanding of programming techniques available to the Prolog programmer, regardless of the particular application.

We hope that we have included enough applications to satisfy most tastes. Naturally, all of the applications deal with areas that suit Prolog's way of representing the world. You will not find how to calculate the flow of heat through a square metal pipe, for example. It is possible to solve such problems using Prolog, but the expressiveness and power of Prolog is not shown to advantage on problems which are essentially repetitious calculations over arrays of numbers. We would like to be able to discuss large Prolog programs, such as those that are used by Artificial Intelligence researchers for understanding natural language. Unfortunately, the aims of a book like this one preclude discussion of programs that are longer than a page of text and which would appeal only to a specialised audience.

7.1 A Sorted Tree Dictionary

Suppose we wish to make associations between items of information, and retrieve them when required. For example, an ordinary dictionary associates a word with its definition, and a foreign language dictionary associates a word in one language with a word in another language. We have already seen one way to make a dictionary: with facts.

If we wanted to make an index of the performance of horses in the British Isles during the year 1938, we could simply define facts \(\text{winnings}(X,Y)\)
where $X$ is the name of the horse, and $Y$ is the number of guineas (a unit of currency) won by the horse. The following database of facts could serve as part of such an index:

```
winnings(abaris, 582).
winnings(careful, 17).
winnings(jingling_silver, 300).
winnings(maloja, 356).
```

If we wanted to find out how much was won by maloja, we would simply ask the right question, and Prolog would give us the answer:

```
?- winnings(maloja, X).
X=356
```

Remember that when Prolog searches through a database to find a matching fact, it starts at the top of the database and works its way down. This means that if our dictionary database is arranged in alphabetical order, as is the one above, then Prolog will take a short amount of time to find the winnings for ablaze, and it will take longer to find the winnings for zoltan. Although Prolog can look through its database much faster than you could look through a printed index, it is silly to search the index from beginning to end if we know that the horse we are looking for is at the end. Also, although Prolog has been designed to search its database quickly, it is not always as fast as we would wish. Depending on how large your index is, and depending on how much information you have stored about each horse, Prolog might take an uncomfortably long amount of time to search the index.

For these reasons and others, computer scientists have devoted much effort to finding good ways to store information such as indices and dictionaries. Prolog itself uses some of these methods to store its own facts and rules, but it is sometimes helpful to use these methods in our programs. We shall describe one such method for representing a dictionary, called the sorted tree. The sorted tree is both an efficient way of using a dictionary, and a demonstration of how lists of structures are helpful.

A sorted tree consists of some structures called nodes, where there is one node for each entry in the dictionary. Each node has four components. It contains the two associated items of information, rather like winnings, above. One of these items, called the key, is the one whose name determines its place in the dictionary (the name of the horse in our example). The other is used to store any other information about the object involved (the winnings in our example). In addition, each node contains a tail (like the tail of a list) to a node containing a key whose name is alphabetically less than the name of the key in the node itself. Furthermore, the node contains