3 Lists, Operators, Arithmetic

In this chapter we will study a special notation for lists, one of the simplest and most useful structures, and some programs for typical operations on lists. We will also look at simple arithmetic and the operator notation which often improves the readability of programs. Basic Prolog of Chapter 2, extended with these three additions, becomes a convenient framework for writing interesting programs.

3.1 Representation of lists

The *list* is a simple data structure widely used in non-numeric programming. A list is a sequence of any number of items, such as **ann**, **tennis**, **tom**, **skiing**. Such a list can be written in Prolog as:

[ann, tennis, tom, skiing]

This is, however, only the external appearance of lists. As we have already seen in Chapter 2, all structured objects in Prolog are trees. Lists are no exception to this.

How can a list be represented as a standard Prolog object? We have to consider two cases: the list is either empty or non-empty. In the first case, the list is simply written as a Prolog atom, []. In the second case, the list can be viewed as consisting of two things:

(1) the first item, called the *head* of the list;
(2) the remaining part of the list, called the *tail*.

For our example list

[ann, tennis, tom, skiing]

the head is **ann** and the tail is the list

[tennis, tom, skiing]

In general, the head can be anything (any Prolog object, for example, a tree or a variable); the tail has to be a list. The head and the tail are then combined into a structure by a special functor. The choice of this functor depends on the Prolog implementation; we will assume here that it is the dot:

.(Head, Tail)

Since **Tail** is in turn a list, it is either empty or it has its own head and tail. Therefore, to represent lists of any length no additional principle is needed. Our example list is then represented as the term:

.(ann, .(tennis, .(tom, .(skiing, []))))

Figure 3.1 shows the corresponding tree structure. Note that the empty list appears in the above term. This is because the one but last tail is a single item list:

[skiing]

This list has the empty list as its tail:

[skiing] = .(skiing, [])

This example shows how the general principle for structuring data objects in Prolog also applies to lists of any length. As our example also shows, the straightforward notation with dots and possibly deep nesting of subterms in the tail part can produce rather confusing expressions. This is the reason why Prolog provides the neater notation for lists, so that they can be written as sequences of items enclosed in square brackets. A programmer can use both notations, but the square bracket notation is, of course, normally preferred. We will be aware, however, that this is only a cosmetic improvement and that our lists will be internally represented as binary trees. When such terms are

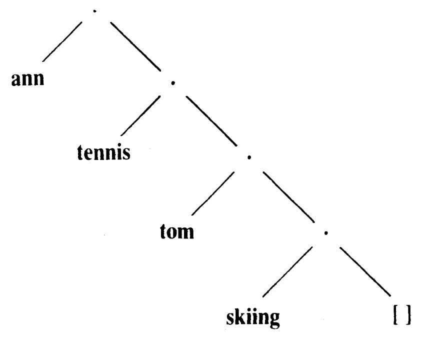

Figure 3.1 Tree representation of the list **[ann, tennis, tom, skiing]**.

output they will be automatically converted into their neater form. Thus the following conversation with Prolog is possible:

```
?- List1 = [a,b,c],
   List2 = .( a, .( b, .( c, [] ) ) ).

List1 = [a,b,c]
List2 = [a,b,c]

?- Hobbies1 = .( tennis, .( music, [] ) ),
   Hobbies2 = [ skiing, food],
   L = [ ann, Hobbies1, tom, Hobbies2].

Hobbies1 = [ tennis, music]
Hobbies2 = [ skiing, food]
L = [ ann, [tennis,music], tom, [skiing,food] ]
```

This example also reminds us that the elements of a list can be objects of any kind, in particular they can also be lists.

It is often practical to treat the whole tail as a single object. For example, let

```
L = [a,b,c]
```

Then we could write

```
Tail = [b,c]   and   L = .( a, Tail)
```

To express this in the square bracket notation for lists, Prolog provides another notational extension, the vertical bar, which separates the head and the tail:

```
L = [ a | Tail]
```

The vertical bar notation is in fact more general: we can list any number of elements followed by '|' and the list of remaining items. Thus alternative ways of writing the above list are:

```
[a,b,c] = [a | [b,c] ] = [a,b | [c] ] = [a,b,c | [] ]
```

To summarize:

- A list is a data structure that is either empty or consists of two parts: a *head* and a *tail*. The tail itself has to be a list.

- Lists are handled in Prolog as a special case of binary trees. For improved readability Prolog provides a special notation for lists, thus accepting lists written as:

```
[ Item1, Item2, ...]
```

or

```
[ Head | Tail]
```

or

```
[ Item1, Item2, ... | Others]
```

3.2 Some operations on lists

Lists can be used to represent sets although there is a difference: the order of elements in a set does not matter while the order of items in a list does; also, the same object can occur repeatedly in a list. Still, the most common operations on lists are similar to those on sets. Among them are:

- checking whether some object is an element of a list, which corresponds to checking for the set membership;
- concatenation of two lists, obtaining a third list, which corresponds to the union of sets;
- adding a new object to a list, or deleting some object from it.

In the remainder of this section we give programs for these and some other operations on lists.

3.2.1 Membership

Let us implement the membership relation as

```
member( X, L)
```

where X is an object and L is a list. The goal **member(X, L)** is true if X occurs in L. For example,

```
member( b, [a,b,c] )
```

is true,

```
member( b, [a,[b,c]] )
```

is not true, but

```
member( [b,c], [a,[b,c]] )
```

is true. The program for the membership relation can be based on the following observation:

X is a member of L if either
(1) X is the head of L, or
(2) X is a member of the tail of L.

This can be written in two clauses, the first is a simple fact and the second is a rule:

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

member( X, [Head | Tail] ) :-
  member( X, Tail).
```

3.2.2 Concatenation

For concatenating lists we will define the relation

```
conc( L1, L2, L3)
```

Here L1 and L2 are two lists, and L3 is their concatenation. For example

```
conc( [a,b], [c,d], [a,b,c,d] )
```

is true, but

```
conc( [a,b], [c,d], [a,b,a,c,d] )
```

is false. In the definition of **conc** we will have again two cases, depending on the first argument, L1:

(1) If the first argument is the empty list then the second and the third arguments must be the same list (call it L); this is expressed by the following Prolog fact:

```
conc( [], L, L).
```

(2) If the first argument of **conc** is a non-empty list then it has a head and a tail and must look like this:

```
[X | L1]
```

Figure 3.2 illustrates the concatenation of **[X | L1]** and some list L2. The result of the concatenation is the list **[X | L3]** where L3 is the concatenation of L1 and L2. In Prolog this is written as:

```
conc( [X | L1], L2, [X | L3] ) :-
  conc( L1, L2, L3).
```

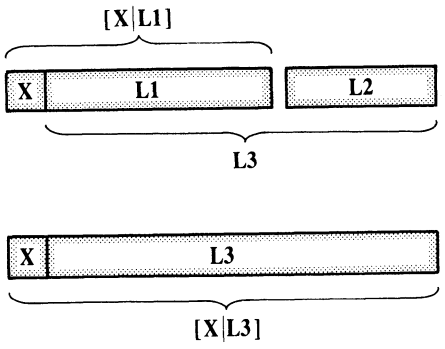

Figure 3.2 Concatenation of lists.

This program can now be used for concatenating given lists, for example:

```
?- conc( [a,b,c], [1,2,3], L).
L = [a,b,c,1,2,3]

?- conc( [a,[b,c],d], [a,[],b], L).
L = [a, [b,c], d, a, [], b]
```

Although the **conc** program looks rather simple it can be used flexibly in many other ways. For example, we can use **conc** in the inverse direction for *decomposing* a given list into two lists, as follows:

```
?- conc( L1, L2, [a,b,c] ).

L1 = []
L2 = [a,b,c];

L1 = [a]
L2 = [b,c];

L1 = [a,b]
L2 = [c];

L1 = [a,b,c]
L2 = [];

no
```

It is possible to decompose the list **[a,b,c]** in four ways, all of which were found by our program through backtracking.

We can also use our program to look for a certain pattern in a list. For example, we can find the months that precede and the months that follow a given month, as in the following goal:

```
?- conc( Before, [may | After],
        [jan,feb,mar,apr,may,jun,jul,aug,sep,oct,nov,dec] ).

Before = [jan,feb,mar,apr]
After = [jun,jul,aug,sep,oct,nov,dec].
```

Further we can find the immediate predecessor and the immediate successor of May by asking:

```
?- conc( _, [Month1,may,Month2 | _],
        [jan,feb,mar,apr,may,jun,jul,aug,sep,oct,nov,dec] ).

Month1 = apr
Month2 = jun
```

Further still, we can, for example, delete from some list, L1, everything that follows three successive occurrences of z in L1 together with the three z's. For example:

```
?- L1 = [a,b,z,z,c,z,z,z,d,e],
   conc( L2, [z,z,z | _], L1).

L1 = [a,b,z,z,c,z,z,z,d,e]
L2 = [a,b,z,z,c]
```

We have already programmed the membership relation. Using **conc**, however, the membership relation could be elegantly programmed by the clause:

```
member1( X, L) :-
  conc( L1, [X | L2], L).
```

This clause says: X is a member of list L if L can be decomposed into two lists so that the second one has X as its head. Of course, **member1** defines the same relation as **member**. We have just used a different name to distinguish between the two implementations. Note that the above clause can be written using anonymous variables as:

```
member1( X, L) :-
  conc( _, [X | _], L).
```

It is interesting to compare both implementations of the membership relation, **member** and **member1**. **member** has a rather straightforward procedural meaning, which is as follows:

To check whether some X is a member of some list L:

(1) first check whether the head of L is equal to X, and then

(2) check whether X is a member of the tail of L.

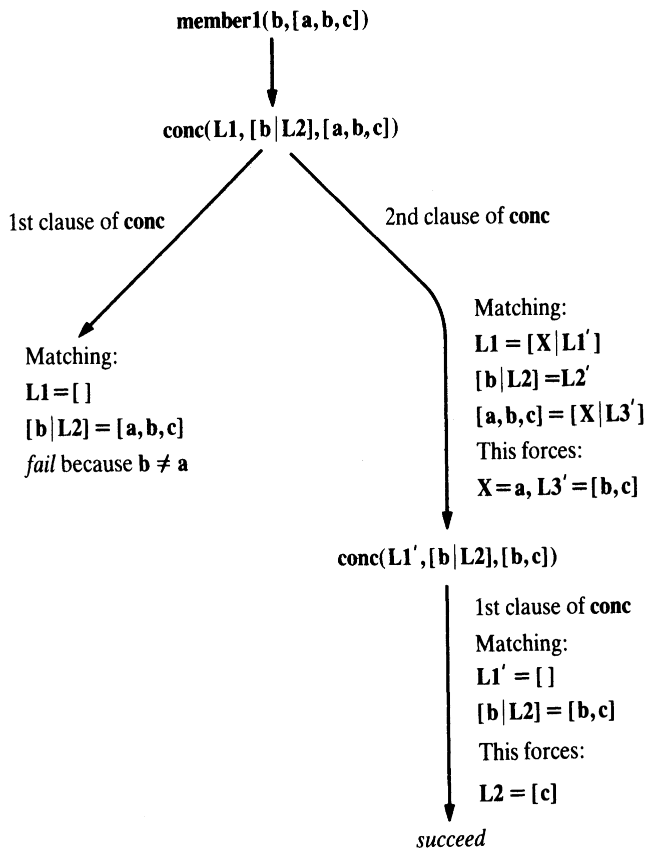

Figure 3.3 Procedure **member1** finds an item in a given list by sequentially searching the list.

On the other hand, the declarative meaning of **member1** is straightforward, but its procedural meaning is not so obvious. An interesting exercise is to find how **member1** actually computes something. An example execution trace will give some idea: let us consider the question:

```
?- member1( b, [a,b,c] ).
```

Figure 3.3 shows the execution trace. From the trace we can infer that **member1** behaves similarly to **member**. It scans the list, element by element, until the item in question is found or the list is exhausted.

Exercises

3.1 (a) Write a goal, using **conc**, to delete the last three elements from a list L producing another list L1. Hint: L is the concatenation of L1 and a three-element list.

(b) Write a sequence of goals to delete the first three elements and the last three elements from a list L producing list L2.

3.2 Define the relation

```
last( Item, List)
```

so that **Item** is the last element of a list **List**. Write two versions: (a) using the **conc** relation, (b) without **conc**.

3.2.3 Adding an item

To add an item to a list, it is easiest to put the new item in front of the list so that it becomes the new head. If X is the new item and the list to which X is added is L then the resulting list is simply

```
[X | L]
```

So we actually need no procedure for adding a new element in front of the list. Nevertheless, if we want to define such a procedure explicitly, it can be written as the fact:

```
add( X, L, [X | L] ).
```

3.2.4 Deleting an item

Deleting an item, X, from a list, L, can be programmed as a relation

```
del( X, L, L1)
```

where L1 is equal to the list L with the item X removed. The **del** relation can be defined similarly to the membership relation. We have, again, two cases:

(1) If X is the head of the list then the result after the deletion is the tail of the list.

(2) If X is in the tail then it is deleted from there.

```
del( X, [X | Tail], Tail).

del( X, [Y | Tail], [Y | Tail1] ) :-
  del( X, Tail, Tail1).
```

Like **member**, **del** is also non-deterministic in nature. If there are several occurrences of X in the list then **del** will be able to delete anyone of them by backtracking. Of course, each alternative execution will only delete one occurrence of X, leaving the others untouched. For example:

```
?- del( a, [a,b,a,a], L).

L = [b,a,a];

L = [a,b,a];

L = [a,b,a];

no
```

del will fail if the list does not contain the item to be deleted.

del can also be used in the inverse direction, to add an item to a list by inserting the new item anywhere in the list. For example, if we want to insert **a** at any place in the list **[1,2,3]** then we can do this by asking the question: What is L such that after deleting **a** from L we obtain **[1,2,3]**?

```
?- del( a, L, [1,2,3] ).

L = [a,1,2,3];

L = [1,a,2,3];

L = [1,2,a,3];

L = [1,2,3,a];

no
```

In general, the operation of inserting X at any place in some list **List** giving **BiggerList** can be defined by the clause:

```
insert( X, List, BiggerList) :-
  del( X, BiggerList, List).
```

In **member1** we elegantly implemented the membership relation by using **conc**. We can also use **del** to test for membership. The idea is simple: some X is a member of **List** if X can be deleted from **List**:

```
member2( X, List) :-
  del( X, List, _).
```

3.2.5 Sublist

Let us now consider the **sublist** relation. This relation has two arguments, a list L and a list S such that S occurs within L as its sublist. So

```
sublist( [c,d,e], [a,b,c,d,e,f] )
```

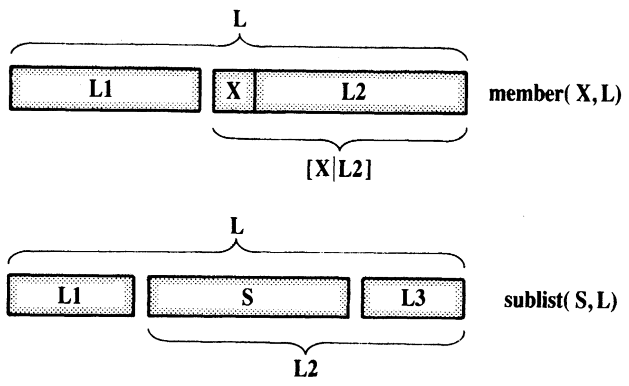

Figure 3.4 The **member** and **sublist** relations.

is true, but

```
sublist( [c,e], [a,b,c,d,e,f] )
```

is not. The Prolog program for **sublist** can be based on the same idea as **member1**, only this time the relation is more general (see Figure 3.4). Accordingly, the relation can be formulated as:

S is a sublist of L if
(1) L can be decomposed into two lists, L1 and L2, and
(2) L2 can be decomposed into two lists, S and some L3.

As we have seen before, the **conc** relation can be used for decomposing lists. So the above formulation can be expressed in Prolog as:

```
sublist( S, L) :-
  conc( L1, L2, L),
  conc( S, L3, L2).
```

Of course, the **sublist** procedure can be used flexibly in several ways. Although it was designed to check if some list occurs as a sublist within another list it can also be used, for example, to find all sublists of a given list:

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

S = [];

S = [a];

S = [a,b];

S = [a,b,c];

S = [b];

...
```

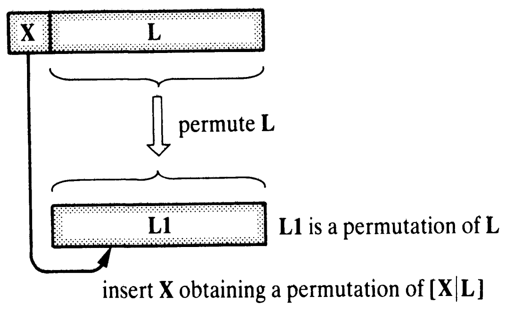

Figure 3.5 One way of constructing a permutation of the list [X | L].

3.2.6 Permutations

Sometimes it is useful to generate permutations of a given list. To this end, we will define the **permutation** relation with two arguments. The arguments are two lists such that one is a permutation of the other. The intention is to generate permutations of a list through backtracking using the **permutation** procedure, as in the following example:

```
?- permutation( [a,b,c], P).

P = [a,b,c];

P = [a,c,b];

P = [b,a,c];

...
```

The program for **permutation** can be, again, based on the consideration of two cases, depending on the first list:

(1) If the first list is empty then the second list must also be empty.

(2) If the first list is not empty then it has the form [X | L], and a permutation of such a list can be constructed as shown in Figure 3.5: first permute L obtaining L1 and then insert X at any position into L1.

Two Prolog clauses that correspond to these two cases are:

```
permutation( [], [] ).

permutation( [X | L], P) :-
  permutation( L, L1),
  insert( X, L1, P).
```

One alternative to this program would be to delete an element, X, from the first list, permute the rest of it obtaining a list P, and then add X in front of P. The corresponding program is:

```
permutation2( [], [] ).

permutation2( L, [X | P] ) :-
  del( X, L, L1),
  permutation2( L1, P).
```

It is instructive to do some experiments with our permutation programs. Its normal use would be something like this:

```
?- permutation( [red,blue,green], P).
```

This would result in all six permutations, as intended:

```
P = [ red, blue, green];

P = [ red, green, blue];

P = [ blue, red, green];

P = [ blue, green, red];

P = [ green, red, blue];

P = [ green, blue, red];

no
```

Another attempt to use **permutation** is:

```
?- permutation( L, [a,b,c] ).
```

Our first version, **permutation**, will now instantiate L successfully to all six permutations. If the user then requests more solutions, the program would never answer 'no' because it would get into an infinite loop trying to find another permutation when there is none. Our second version, **permutation2**, will in this case find only the first (identical) permutation and then immediately get into an infinite loop. Thus, some care is necessary when using these **permutation** relations.

Exercises

3.3 Define two predicates

evenlength(List) and **oddlength(List)**

so that they are true if their argument is a list of even or odd length respectively. For example, the list **[a,b,c,d]** is 'evenlength' and **[a,b,c]** is 'oddlength'.

3.4 Define the relation

```
reverse( List, ReversedList)
```

that reverses lists. For example, **reverse([a,b,c,d], [d,c,b,a])**.

3.5 Define the predicate **palindrome(List)**. A list is a palindrome if it reads the same in the forward and in the backward direction. For example, **[m,a,d,a,m]**.

3.6 Define the relation

```
shift( List1, List2)
```

so that **List2** is **List1** 'shifted rotationally' by one element to the left. For example,

```
?- shift( [1,2,3,4,5], L1),
   shift( L1, L2).
```

produces:

```
L1 = [2,3,4,5,1]
L2 = [3,4,5,1,2]
```

3.7 Define the relation

```
translate( List1, List2)
```

to translate a list of numbers between 0 and 9 to a list of the corresponding words. For example:

```
translate( [3,5,1,3], [three,five,one,three] )
```

Use the following as an auxiliary relation:

```
means( 0, zero).  means( 1, one).  means( 2, two).  ...
```

3.8 Define the relation

```
subset( Set, Subset)
```

where **Set** and **Subset** are two lists representing two sets. We would like to be able to use this relation not only to check for the subset relation, but also to generate all possible subsets of a given set. For example:

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

S = [a,b,c];

S = [b,c];

S = [c];
```

```
S = [];
S = [a,c];
S = [a];
...
```

3.9 Define the relation

dividelist(List, List1, List2)

so that the elements of **List** are partitioned between **List1** and **List2**, and **List1** and **List2** are of approximately the same length. For example, **partition([a,b,c,d,e], [a,c,e], [b,d])**.

3.10 Rewrite the monkey and banana program of Chapter 2 as the relation

canget(State, Actions)

to answer not just 'yes' or 'no', but to produce a sequence of monkey's actions that lead to success. Let **Actions** be such a sequence represented as a list of moves:

```
Actions = [ walk(door,window), push(window,middle), climb, grasp]
```

3.11 Define the relation

flatten(List, FlatList)

where **List** can be a list of lists, and **FlatList** is **List** 'flattened' so that the elements of **List**'s sublists (or sub-sublists) are reorganized as one plain list. For example:

```
?- flatten( [a,b,[c,d],[],[[[e]]],f], L).
L = [a,b,c,d,e,f]
```

3.3 Operator notation

In mathematics we are used to writing expressions like

$$2*a + b*c$$

where + and * are operators, and 2, *a, b, c* are arguments. In particular, + and * are said to be *infix* operators because they appear *between* the two arguments. Such expressions can be represented as trees, as in Figure 3.6, and can be written as Prolog terms with + and * as functors:

```
+( *(2,a), *(b,c) )
```

Since we would normally prefer to have such expressions written in the usual,