

Chapter 3

The list structure

List, as we have seen in previous courses, is one of the simplest, but very useful, structures. It is basically a sequence of items, such as {ann, tennis, tom, skiing}, which is represented as [ann, tennis, tom, skiing] in Prolog.

More deeply, a list is represented as a tree, just like any standard Prolog object. A list is either empty, i.e., []; or consisting of two parts: the first item, the *head*, and the rest of the list, or the *tail*. Thus, for the above list, ann is the head, and [tennis, tom, skiing] is the tail.

The dirty detail

In general, the head of a list can be anything, and the tail has to be a list. They are then combined into a list via a special functor, `'.'`: `.(Head, Tail)`. Since tail is again a list, it is really a recursive structure.

For example, the above list is really represented as `.(ann,.(tennis,.(tom,.(skiing,[])))`.

But, we will always represent the above list as

```
[ann, tennis, tom, skiing]
```

A cleaner expression

Although any list is *internally* represented as a tree, when it is output, it is automatically converted to a cleaner form. Below is an example of a conversation with Prolog.

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

```
list1=[a,b,b]
```

```
?-Hobbies1=[tennis,food],
```

```
Hobbies2=[skiing,music],
```

```
L=[ann,Hobbies1,tom,Hobbies2].
```

```
Hobbies1=[tennis,food]
```

```
Hobbies2=[skiing,music]
```

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

Thus, a list can be a collection of anything, including other lists.

Yet another way

It is only necessary to treat the whole tail as a single object. For instance, if $L=[a,b,c]$, we could write $L=(a,Tail)$ and $Tail=[b,c]$. Actually, Prolog allows us to write $L=[a|Tail]$.

Then, the following makes sense:

$$[a,b,c]=[a|[b,c]]=[a,b|[c]]=[a,b,c|[]].$$

Now, we are ready to have some fun. We will see how to check if an item is a *member* of a list; how to *concatenate* two lists; how to *add* an item into a list; and how to *delete* an item from a list.

These are examples of list processing operations.

Membership

Let X be an object, and L be a list, the relation $\text{member}(X,L)$ holds iff X is *an item of* L .

For example, $\text{member}(b, [a,b,b])$ is true, while $\text{member}([b,c], [a,b,c])$ is false.

As we discussed before, in general, X is a member of L either X is the head of L ; or X is a member of the tail of L . Thus,

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

For example,

```
6 ?- member(1, [1, 2, 3]).
```

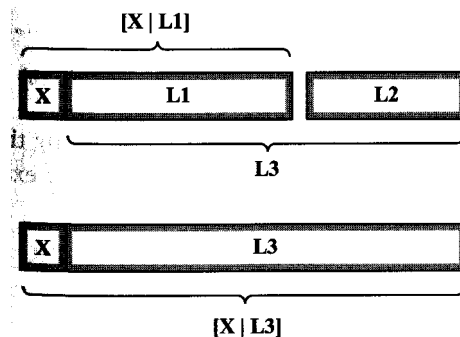
Yes

Concatenation

Let $L1$, $L2$ and $L3$ be lists, $\text{conc}(L1,L2,L3)$ is true iff $L3$ is *the concatenation of* $L1$ and $L2$. For example, $\text{conc}([a,b],[c,d],[a,b,c,d])$ is true, but $\text{conc}([a,b],[c],[a,b,c,d])$ is false.

There are again two cases, depending on the nature of $L1$. If it is empty, $L3$ is the same as $L2$. In general, the concatenation of $[X|L1]$ and $L2$ must be $[X|L3]$ if the concatenation of $L1$ and $L2$ is $L3$. We have the following Prolog program.

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



Despite the innocent looking

1. A rather direct one

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

2. Decomposition

?-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

3. Pattern recognition

```
?-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].
```

Question: What does the following do?

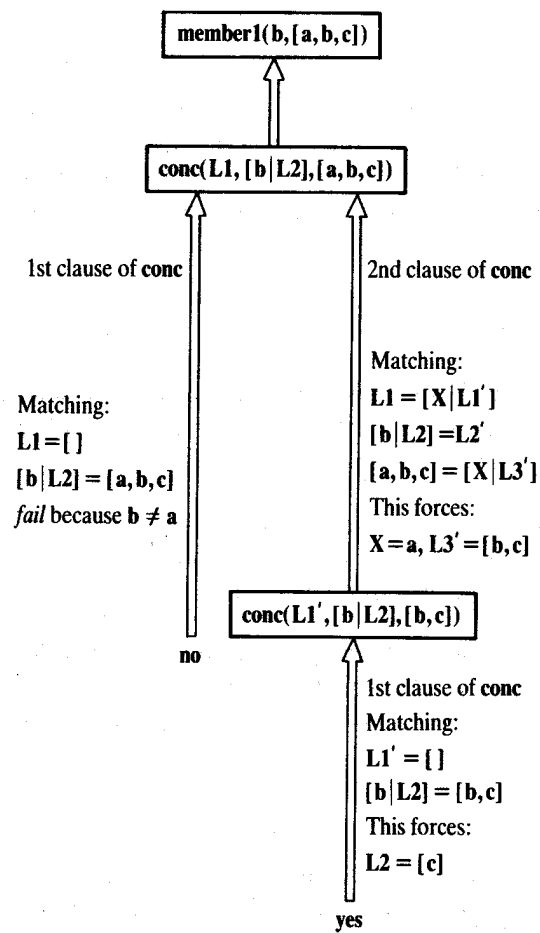
```
?- L1=[a, b, z, z, c, z, z, z, d, e],  
conc(L2, [z, z, z|_], L1).
```

Question: What does the following mean?

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

Trace it through

The following traces through what happens to `?-member1(b, [a,b,c])`.



Homework: Exercises 3.1. and 3.2.

Add and delete

When adding an item in to a list, it is rather easy to put it at the head (Stack!) Thus,

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

It is a bit more complex to delete X from L. Again, there could be two cases: If X is at the head of L, it is a piece of cake: simply cut it off. Otherwise, we recursively delete it from the tail of L. Thus,

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

For example,

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

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

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

```
no
```

Insert vs. delete

The `delete` relation can also be used inversely: If we obtain `L2` by deleting `X` from `L1`, then we can get `L1` by inserting `X` into `L2`. For example,

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

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

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

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

```
no
```

Simply put,

```
insert(X,List,BigList):-delete(X,BigList,List).
```

Question: What does the following mean?

```
member2(X,L):-delete(X,L,_).
```

Sublist

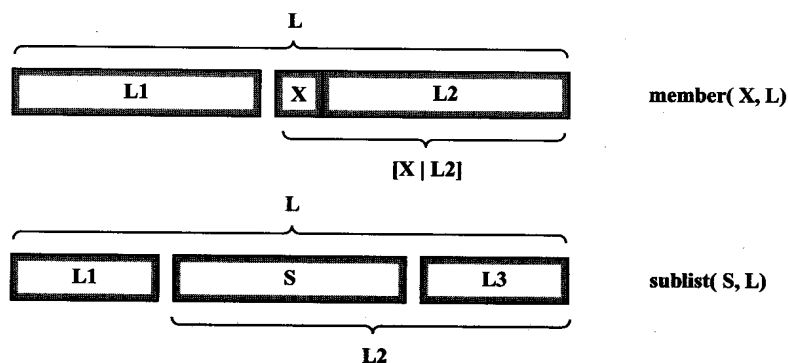
Let $L1$ and $L2$ be two lists, $sublist(L1,L2)$ holds if $L1$ is a subsequence of $L2$.

For example, $sublist([a,b],[c,a,b,d])$ is true, while $sublist([a,b],[a,c,b])$ is false.

Question: How to write this up in Prolog?

Answer: $L2$ must be the concatenation of three pieces, $_$, $L1$, and $_$. Thus,

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



or

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

Permutation

Let L be a list, `permutation(L,P)` holds if P is a permutation of L . If $|L|=n$, there are $n!$ possible P .

Question: How did we do it in Data Structure?

It is easy to do it in Prolog. Again, two cases, based on the nature of L . If it is empty, P must be empty too. Otherwise, we generate a permutation of its tail, then insert its head to generate a permutation of the whole list. Hence,

```
permutation1([], []).
permutation1([X|L],P):-
    permutation1(L,L1),add(X,L1,P).
```

Thinking about the relationship between `delete` and `insert`, we can have the following alternative code:

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

Question: If you play with the above two versions, you find out that only the second one works. Why? Modify the `add` definition so that `permutation1` works as well.

Homework: Since all the exercises at the end of §3.2 are so good, and, easy, do them all.

Notation for operators

When we write down an arithmetic notation such as $2 * a + b * c$, Prolog will represent it internally as $+(*(2, a), *(b, c))$, which can be further represented in a tree structure.

We have to provide further information on the *precedence* of the involved operators for Prolog to correctly represent an expression such as $a + b * c$.

The general rule is that the one with the highest precedence will be the *principal functor*. Since the above is normally understood as $a + (b * c)$, we will say that '+' has a higher precedence than '*'. Thus, it will be internally represented as $+(a, *(b, c))$.

Do it yourself

When programming in Prolog, we need to define our own operators. For instance, we can define `has` as an infix operator so that the fact *peter has information* is equivalent to the following Prolog fact

```
has(peter,information).
```

Technically speaking, we can use the following *directive* to provide some *syntactic* information

```
:-op(600,xfx,has).
```

to tell Prolog that 'has' is an infix operator with a precedence of 600.

Technicality

To properly define an operator, we have to specify two things.

1. Is it a prefix, infix, or postfix operator?

This is easy, if it is infix, you use either xfx , xfy or yfx . If it is pre(post)fix, you use fx , $fy(xf, yf)$.

2. Which one is to use?

This depends on the precedence of the potential arguments. The precedence of an unstructured, or parenthesesized, term is 0. If it is a structure, then its precedence equals to that of its principal operator. Moreover, x represents an argument whose precedence must be strictly lower than that of f ; and y represents an argument whose precedence could be either strictly lower than, or equal to, that of f .

Examples

1. The normal binary $-$ operator should be declared as a 'yfx' type. Since, we interpret $a - b - c$ as $(a - b) - c$, in which the precedence of $a - b$ is the same as that of $-$. (?) Thus, we *cannot* use 'x' for the first argument.

2. The logic *not* operator is a prefix one, used in `not p`, meaning the negation of `p`. Thus, we have to define it as either 'fx', or 'fy'. If we declare it using 'fx'. Then, such expression as `not not p` will be illegal, since the precedence of `not p`, as a structured object, has the same precedence as that of *not*. Thus, we have to write such expression as `not (not p)`. In this case, if we use 'fy', it will be *convenient*.

3. One of the de Morgan's rule is as follows:

$$\neg(A \wedge B) \Leftrightarrow \neg A \vee \neg B.$$

One way to represent this piece of knowledge is to use the following clause:

`equivalent(not(and(A,B)), or(not(A),not(B))).`

A much better way is to declare the following operators:

`:-op(800,xfx,<===>).`

`:-op(700,xfy,v).`

`:-op(600,xfy,&).`

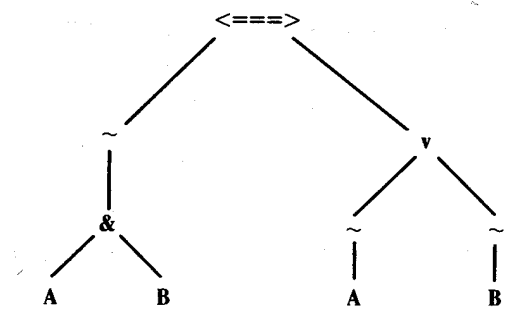
`:-op(500,fy,~).`

`~(A & B)<===>~A v ~B.`

Question: Why does the above work?

Just a point

The precedence is an integer between 1 and 1200. The operator with the highest precedence in an expression is the principle functor of that expression. Operators with lower precedence bind stronger. Below shows the structure of the above fact.



Homework: Exercises 3.12, 3.14(a,c), and 3.15.

Arithmetic operations

We can use the normal arithmetic operations in a Prolog program. For example

```
?-X is 3-2.
```

sends back `X is 1`.

The following is a program to find out the *greatest common divisor* of `X` and `Y`.

```
gcd(X,X,X).  
gcd(X,Y,D):-X<Y,Y1 is Y-X,gcd(X,Y1,D).  
gcd(X,Y,D):-X>Y,gcd(Y,X,D).
```

The following finds out the length of `L`.

```
length([],0).  
length(_|Tail,N):-length(Tail,N1),N is N1+1.
```

Homework: Exercises 3.16, 3.17, and 3.19.