Low-level list processing

A list can be specified in terms of a head and a tail.

The list `[1,2,3]` can be specified as this:

```
[1 | [2, 3]]
```

More generally, a list can be described as a sequence of initial elements and a tail.

The list `[1,2,3,4]` can be specified in any of these ways:

```
[1 | [2,3,4]]

[1,2 | [3,4]]

[1,2,3 | [4]]

[1,2,3,4 | []]
```


Low-level list processing, continued

Consider this unification:

```
?- [H|T] = [1,2,3,4].
H = 1
T = [2, 3, 4]
```

What instantiations are produced by these unifications?

```
?- [X, Y | T] = [1, 2, 3].


?- [X, Y | T] = [1, 2].


?- [1, 2 | [3,4]] = [H | T].


?- A = [1], B = [A|A].
```

How does list construction and unification in Prolog compare to ML?

Writing list predicates, continued

A rule to produce the head of a list:

```
head(L, H) :- L = [H|_].
```

The head of L is H if L unifies with a list whose head is H.

Usage:

```
?- head([1,2,3],H).
H = 1

?- head([], X).
No

?- L = [X,X,b,c], head(L, a).
L = [a, a, b, c]
X = a
```

Problem: Define `head/2` more concisely.

Writing list predicates, continued

Here is one way to implement the standard `member/2` predicate:

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

Usage:

```
?- member(1, [2,1,4,5]).
Yes

?- member(a, [2,1,4,5]).
No
```

We've seen that `member` can be used to generate the elements in a list:

```
?- member(X, [a,b,c]).
X = a ;
X = b ;
...
```

How does that generation work?

Problem: Define `member/2` more concisely.

Writing list predicates, continued

A more concise definition of `member/2`:

```
member(X,[X|_]).
```
X is a member of the list having X as its head

```
member(X,[_|T]) :- member(X,T).
```
X is a member of the list having T as its tail if X is a member of T

Problem: Define a predicate `last(L,X)` that describes the relationship between a list L and its last element, X.

```
?- last([a,b,c], X).
X = c
Yes

?- last([], X).
No

?- last(L,last), head(L,first), length(L,2).
L = [first, last]
Yes
```


Writing list predicates, continued

`last` is a built-in predicate but here is one way to write it:

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

Problem: Write a predicate `len/2` that behaves like the built-in `length/2`

```
?- len([], N).
N = 0

?- len([a,b,c,d], N).
N = 4

?- len([a,b,c,d], 5).
No

?- len(L,1).
L = [_G295]
```


Writing list predicates, continued

Here is `len`:

```
len([], 0).
len([_|T],Len) :- len(T,TLen), Len is TLen + 1.
```

Problem: Define a predicate `allsame(L)` that describes lists in which all elements have the same value.

```
?- allsame([a,a,a]).
Yes

?- allsame([a,b,a]).
No

?- allsame([A,B,C]), B = 1.
A = 1
B = 1
C = 1

?- length(L,5), allsame(L), head(L,x).  % Any order works!
L = [x, x, x, x, x]
```


Writing list predicates, continued

Here's another way to test `allsame`:

```
?- allsame(L).

L = [_G305] ;

L = [_G305, _G305] ;

L = [_G305, _G305, _G305] ;

L = [_G305, _G305, _G305, _G305] ;

L = [_G305, _G305, _G305, _G305, _G305] ;
```

Could we test `member` in a similar way? How about `append`?

Writing list predicates, continued

Problem: Define a predicate `listeq(L1,L2)` that describes the relationship between two lists `L1` and `L2` that hold the same sequence of values.

Problem: Define a predicate `p/1` that behaves like this:

```
?- p(X).
X = [] ;
X = [_G272] ;
X = [_G272, _G272] ;
X = [_G272, _G272, _G278] ;
X = [_G272, _G272, _G278, _G278] ;
X = [_G272, _G272, _G278, _G278, _G284] ;
```


The `append` predicate

Recall the description of the built-in append predicate:

`append(?List1, ?List2, ?List3)`
Succeeds when `List3` unifies with the concatenation of `List1` and `List2`.

The usual definition of `append`:

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

How does it work?

Try tracing it. To avoid getting the built-in version, define the above as `my_append` instead of `append` (three places). Then try these:

```
?- gtrace, my_append([1,2,3,4],[a,b,c,d],X).

?- gtrace, my_append([a,b,c,d,e,f,g],[],X).
```


Lists are structures

In fact, lists are structures:

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

Essentially, `./2` is the "cons" operation in Prolog.

By default, lists are shown using the `[...]` notation:

```
?- X = .(a, .(b,[])).
X = [a, b]
```

We can write `member/2` like this:

```
member(X, .(X,_)).
member(X, .(_,T)) :- member(X,T).
```

Speculate: What the following produce?

```
?- X = .(1,1).
```