Deep Lists

1 Theoretical Considerations

In this lesson, we will present a generalization of the list type, where the elements of a list can be lists themselves. In this way, a deep list can contain list elements at different levels of imbrication.

Examples:

```
L1 = [1,2,3,[4]]
L2 = [[1],[2],[3],[4,5]]
L3 = [[],2,3,4,[5,[6]],[7]]
L4 = [[[[1]]],1,[1]]
L5 = [1,[2],[[3]],[[[4]]],[5,[6,[7,[8,[9],10],11],12],13]]
L6 = [alpha,2,[beta],[gamma,[8]]]
```

A deep list (DL) in prolog represents a recursive structure, where several lists of variable depth are nested one into another. Formally, a deep list can be defined as:

$$DL = [H|T] \ where \ H \in \{atom, list, deep\ list\} \ and \ T = deep\ list$$

Remember. A simple (or shallow) list is a trivial case of a deep list.

All operations defined for shallow lists can also be used with deep lists including (but not only) the studied *member/2*, *append/3* and *delete/3* predicates. To understand how operations on deep lists work, one can consider a deep list equivalent to a shallow list with different types of elements, but only those on the first level.

Test the following queries:

Query
?- member(2,L5).
?- member([2], L5).
?- member(X, L5).
?- append(L1,R,L2).
?- append(L4,L5,R).
?- delete(1, L4,R).
?- delete(13,L5,R).

1.1 The „atomic” predicate

To execute different operations on all elements of a deep list we have to know how to treat these elements according to their type (if they are atoms we process them, if they are lists further decomposition might be needed before processing).

We will use the built-in predicate *atomic(X)* to check if X is a simple element (a number or a symbol) or a complex structure (for example: another list).

Test the following queries:

Query
```
? - atomic(apple).
? - atomic(4).
? - atomic(X).
? - atomic(apple(2)).
? - atomic([1,2,3]).
? - atomic([]).
```

1.2 The „depth” predicate

This predicate counts the maximum level of imbrication of a deep list. The depth of an atom is defined as 0, and the depth of a shallow list (including the empty list) as 1.

The predicate calculates the maximum depth of a deep list, it iterates over the list elements and checks their nature (using the built-in predicate *atomic/1*). If the element is atomic, it calls the predicate on its tail (clause 2), while if it is not atomic then it also calls recursively on that element too (clause 2 fails and backtracks to clause 3). The depth of a list is equal to the maximum depth of an element plus one.

Code
```
max(A, B, A):- A>B, !.
max(_, B, B).

depth([],1).
depth([H|T],R):- atomic(H), !, depth(T,R).
depth([H|T],R):- depth(H,R1), depth(T,R2), R3 is R1+1, max(R3,R2,R).
```

When computing the maximum depth, we will have three branches:

1. We arrived at the empty list. The depth is 1. **(clause 1)**
2. We have an atomic head, we ignore it, since it doesn’t influence the depth. The depth of the list will be equal to the depth of the tail. Thus, we must call recursively on the tail, to continue the traversal of the list. **(clause 2)**
3. We have a list in the head of the deep list. In this case the depth of the list will be either the depth of the tail, or the depth of the head increased by one (*Why?*). **(clause 3)**

Test the predicate for lists L1-L6 presented above, for example:

Query
```
?- trace, depth(L1, R).
```

1.3 The „flatten” predicate

This predicate flattens a deep list. The resulted list will contain only atomic elements (the equivalent shallow list of said deep list, containing all the elements, but with nesting level 1). In this case as well, we need to verify the nature of each element. If the element is a list, it will also need to be flattened.

This operation means obtaining a simple list. In order to do this, we take only the atomic elements from the source list and place them in the result.

Code
```
flatten([],[]).
flatten([H|T], [H|R]):- atomic(H), !, flatten(T,R).
flatten([H|T], R):- flatten(H,R1), flatten(T,R2), append(R1,R2,R).
```

We have 3 main cases:

- We have to deal with the empty list. Flattening the empty list results in an empty list. **(clause 1)**
- If the first element of the list is atomic, we put it into the result, and process the rest of the list. **(clause 2)**
- If the first element is not atomic, the result will be composed of all the atomic elements (or flattening) of the head, and all the atomic elements (or flattening) of the tail. *(How do we collect the two results in a single list?)* **(clause 3)**

Note. It is hard to see where in this predicate the flatten is happening. It is actually done by the *append* predicate and by its functionality. For example:

```
L = [[1],2,3,4] where H=[1] and T = [2,3,4]
```

When we append the head and the tail, we get the following:

Query
```
?- append(H,T,R).
R = [1,2,3,4]
```

Try to modify the append in flatten to:

```
append([R1],R2,R).
```

Test the predicate for lists L1-L6 presented above, for example:

Query
```
?- trace, flatten(L1, R).
```

1.4 The „heads” predicate

This predicate extracts all atomic elements at the head of each list.

We will use an auxiliary predicate, which skips all atomic elements of a list. We will take only the first element from each list, then use our auxiliary predicate to skip all the rest of atomic elements.

Code

```
% Variant 1
skip([],[]).
skip([H|T],R):- atomic(H),!,skip(T,R).
skip([H|T],[H|R]):- skip(T,R).

% takes the first element from each list,
% then skips the rest of atomic elements, calls recursively for lists.
heads1([],[]).
heads1([H|T],[H|R]):- atomic(H),!,skip(T,T1), heads1(T1,R).
heads1([H|T],R):- heads1(H,R1), heads1(T,R2),append(R1,R2,R).
```

For the second implementation variant, we will use a third argument (*flag*) to remember if we extract the head of the current list or not.

Code

```
% Variant 2

heads2([],[],_).

% if flag=1, it means that we are at the beginning of the list and we can
% extract the head, but on the recursive call we must set the flag to 0
heads2([H|T],[H|R],1):- atomic(H), !, heads2(T,R,0).

% if flag=0, then we are not at the first atomic element
% and we must continue with the rest of the elements
heads2([H|T],R,0):- atomic(H), !, heads2(T,R,0).

% if we reach this clause it means that the first element is not
% atomic and we must call recursively on this element as well
heads2([H|T],R,_):- heads2(H,R1,1), heads2(T,R2,0), append(R1,R2,R).

% wrapper for heads predicate
heads2(L,R):- heads2(L, R, 1).
```

Test the predicate for lists L1-L6 presented above, for example:

Query

```
?- trace, heads1(L1, R).
?- trace, heads2(L1, R).
```

1.5 The „member" predicate

We can use predicates that we have already learned for a simple implementation of the *member* predicate for deep lists in the following manner:

Code

```
% Variant 1
member1(X, L):- flatten(L,L1), member(X,L1).
```

Observation: This version may only find atomic elements. Check the second variant of the member predicate for a more general solution.

However, if we want to extend the member predicate implemented in previous laboratory session to work on deep lists as well, we must modify the *member/2* predicate in the following manner:

Code

```
% Variant 2
member2(H, [H|_]).
member2(X, [H|_]):- member2(X,H). % H is a list
member2(X, [_|T]):- member2(X,T).
```

Works similarly to the member predicate in the case of the shallow lists, considers as member all elements appearing in the list, atomic or not, at any level.

Test the following queries:

Query

```
?- trace, member2(1,L1).
?- trace, member2(4,L2).
?- trace, member2([5,[6]], L3).
?- trace, member2(X,L4).
?- trace, member2(X,L6).
?- trace, member2(14,L5).
```

2 Exercises

1. Write the *count_atomic(L,R)* predicate that counts the number of atomic elements in list *L* (*all atomic elements on all depths*).

```
Query
?- count_atomic([1,[2],[[3]],[[[4]]],[5,[6,[7,[8,[9],10],11],12],13]], R).
R = 13;
false.
```

2. Write the *sum_atomic(L,R)* predicate that counts the sum of atomic elements from list *L* (*all atomic elements on all depths*).

```
Query
?- sum_atomic([1,[2],[[3]],[[[4]]]], R).
R = 10;
false.
```

3. Write the *replace(X,Y,L,R)* predicate that replaces each occurrence of *X* with *Y* in a deep list *L* (at any imbrication level) and inserts the result into R.

```
Query
?- replace(1, a, [[[[1,2], 3, 1], 4],1,2,[1,7,[[1]]]], R).
R = [[[[a, 2], 3, a], 4], a, 2, [a, 7, [[a]]]];
false.
```

4. Write the *lasts(L,R)* predicate that extracts the last position (immediately before *']'*) of each sublist in *L*.

```
Query
?- lasts([1,2,[3],[4,5],[6,[7,[9,10],8]]], R).
R = [3,5,10,8] ;
false.
```

5. Replace each constant depth sequence in a deep list with its length (do **not** use the *length/2* predicate).

```
Query
? -  len_con_depth([[1,2,3],[2],[2,[2,3,1],5],3,1],R).
R = [[3],[1],[1,[3],1],2].
```

6. Replace each constant depth sequence in a deep list with its depth (do **not** use the *depth/2* predicate).

Query
```
? -  depth_con_depth([[1,2,3],[2],[2,[2,3,1],5],3,1],R).
R = [[1], [1], [1, [2], 1], 0].
```

7. Modify the *member2/2* predicate for deep lists so that it is a deterministic predicate (in this case, deterministic means that only one answer will be returned – there is no *next* answer – multiple tests have to be made on the *member_deterministic/2* predicate with various inputs to verify).

Query
```
?- member_deterministic(1, [1,2,1]).
true.

?- member_deterministic(1, [[1],2,1]).
true.
```

8. Write a predicate that sorts a deep list based on the depth of each element. If two elements have the same depth, then they will be compared based on the atomic elements they contain. (You can use a sorting method of your choice).

Hint:

- L1< L2, if L1 and L2 are lists, or deep lists, and the depth of L1 is smaller than the depth of L2.
- L1 < L2, if L1 and L2 are lists, or deep lists with equal depth, all the elements up to the *k-th* are equal, and the *k+1-th* element of L1 is smaller than the *k+1th* element of L2 (at equal depths, the list with the smaller index of the sublist that gives the last depth is considered larger – as shown in the example for the comparison between 5 and [5])

Query
```
?- sort_depth([[[[1]]], 2, [5,[4],7], [[5],4], [5,[0,9]]], R).
R = [2, [5,[0,9]], [[5],4], [5,[4],7], [[[1]]]] ;
false
```