# Incomplete Structures (Lists & Trees)

## 1 Theoretical Considerations

In this lesson, we will present a special case of structures. Incomplete structures, instead of having a constant element at the end (e.g., as [] for lists or nil for trees), they end in a free variable "_".

### 1.1 Representation

Incomplete structures (partially instantiated structures) offer the possibility of altering the free variable at the end (instantiate it partially), and have the result in the same structure (concatenation at the end does not require an extra output argument). In order to reach the free variable at the end, incomplete structures are traversed in the same manner as complete structures.

However, the clauses which specify the behavior when reaching the end must be explicitly stated (*even in the case of failure!*). Therefore, at the stopping condition, we must check if a free variable has been reached (through the use of the built-in predicate *var/1*). Furthermore, these clauses must be placed in front of all the other predicate clauses – because the free variable at the end will unify with anything. To avoid undesired unifications, those cases must be treated first.

Examples:

| Query |
| --- |
| ?- L = [a, b, c\|_]. |
| ?- L = [1, 2, 3\|T], T = [4, 5\|U]. |
| ?- T = t(7, t(5, t(3, _, _), _), t(11, _, _)). |
| ?- T = t(7, t(5, t(3, A, B), C), t(11, D, E)), D = t(9, F, G). |

### 1.2 Incomplete Lists

Incomplete lists are a special type of incomplete structures. Instead of ending in [], an incomplete list has a free variable as its tail.

Incomplete lists are traversed in the same way as complete lists, using the [H|T] pattern; the difference becomes apparent at the end of the list – where we meet not the empty list *[]* at the end, but a free variable. This has the following implications on the predicates on incomplete lists:

1. testing for the end of the list must ***always*** be performed, even for fail situations – and the fail must be explicit
2. When testing for the end of the list, we must check if we have reached the free variable at the end:

```
some_predicate(L, ...):-var(L), ...
```

3. the clause which checks for the end of the list must ***always*** be the first clause of the predicate (*Why?*)
4. you may add any list at the end of an incomplete list, without needing a separate output structure (adding at the end of an incomplete list can be performed in the same input structure):

   e.g. ?- L = [1, 2, 3|T], T = [4, 5|U], U=[6|_]

Keeping these observations in mind, we will continue with transforming a number of well known list predicates such that they work on incomplete lists.

### 1.2.1 The „member" predicate

Before writing the new predicate, let us follow the execution of the old *member* predicate:

Query

```
?- trace, L = [1, 2, 3|_], member1(3, L).
?- trace, L = [1, 2, 3|_], member1(4, L).
?- trace, L = [1, 2, 3|_], member1(X, L).
```

As you have observed, when the element appears in the list, the predicate member1/2 behaves correctly; but when the element is not a member of the list, instead of answering no, the predicate adds the element at the end of the incomplete list – incorrect behavior. In order to correct this behavior, we need to add a clause which specifies the explicit fail when the end of the list (the free variable) is reached:

Code

```
% must test explicitly for the end of the list
% which means we did not find the element, so we call fail
member_il(_, L):-var(L), !, fail.
% these 2 clauses are the same as for the member1 predicate
member_il(X, [X|_]):-!.
member_il(X, [_|T]):-member_il(X, T).
```

Follow the execution of the previous queries on the new predicate.

### 1.2.2 The „insert" predicate

As you may have already observed, adding an element at the end of an incomplete list doesn't require an additional output argument – the addition may be performed in the input structure.

To do that, we need to traverse the input list element by element and when the end of the list is found, simply modify that free variable such that it contains the new element. If the element is already in the list, don't add it:

Code

```
% found end of list, add element
insert_il1(X, L):-var(L), !, L=[X|_].
insert_il1(X, [X|_]):-!. % element found, stop
insert_il1(X, [_|T]):- insert_il1(X, T). %traverse input list to reach end
```

*Note*. There is a similarity between *insert_il* and *member_il* – the only thing that differs is the functionality when the end of the list is reached.

Also, the *insert_il* and *member1* predicates are very similar. Actually, if we compare their traces more closely, we find that they provide the same behavior! This means that testing for membership on complete lists is equivalent to inserting a new element in an incomplete list – i.e. we can remove the first clause of the *insert_il* predicate (Why?).

Code

```
insert_il2(X, [X|_]):-!.
insert_il2(X, [_|T]):- insert_il2(X, T).
```

The functionality of the first clause can be achieved by the second one as well. If the searched element is not found, it means that we have reached the end of the list. In this case, the free variabile from the end of the list will unify with [X|_], which means that the searched element is inserted at the end of the list.

Follow the execution of:

Query

```
?- trace, L = [1, 2, 3|_], insert_il2(3, L).
?- trace, L = [1, 2, 3|_], insert_il2(4, L).
?- trace, L = [1, 2, 3|_], insert_il2(X, L).
```

### 1.2.3 The „delete" predicate

In the *delete/3* predicate will keep the free variable at the end for the input and output lists. The rest of the predicate will remain as its counterpart for complete lists.

Code

```
delete_il(_, L, L):-var(L), !. % reached end, stop
delete_il(X, [X|T], T):- !.    % found, remove the first appearance and stop
delete_il(X, [H|T], [H|R]):- delete_il(X, T, R). % traverse, search for element
```

*Note*. The stopping condition, which corresponds to reaching the end of the input list, is the first clause and has the known form. Clauses 2 and 3 are the same as for the *delete/3* predicate for complete lists.

Follow the execution of:

Query

```
?- trace, L = [1, 2, 3|_], delete_il(2, L, R).
?- trace, L = [1, 2, 3|_], delete_il(4, L, R).
?- trace, L = [1, 2, 3|_], delete_il(X, L, R).
```

## 1.3 Incomplete Trees

Incomplete trees are another special type of incomplete structures – a branch no longer ends in nil, but in an unbound (free) variable.

The observations made for writing predicates on incomplete lists apply in the case of incomplete trees as well. Thus, we shall apply those observations to develop the same predicates we discussed for lists in the previous section: *search_it/2, insert_it/2, delete_it/3* – for incomplete binary search trees.

### 1.3.1 The „search" predicate

Just like in the case of lists, the predicate which searches for a key in a complete tree does not perform entirely well on incomplete trees – we need to explicitly add a clause for fail situations (for when the free variable is reached):

Code

```
search_it(_, T):- var(T), !, fail.
search_it(Key, t(Key, _, _)):- !.
search_it(Key, t(K, L, _)):- Key<K, !, search_it(Key, L).
search_it(Key, t(_, _, R)):- search_it(Key, R).
```

Follow the execution of:

Query

```
?- trace, T=t(7, t(5, t(3,_,_), t(6,_,_)), t(11,_,_)), search_it(6, T).
?- trace, T=t(7, t(5, t(3,_,_), _), t(11,_,_)), search_it(9, T).
```

### 1.3.2 The „insert" predicate

Since insertion in a binary search tree is performed at the leaf level (i.e. at the end of the structure), we can insert a new key in an incomplete binary search tree, without needing an extra output argument. If we turn again to the analogy with incomplete lists, we find that the predicate which performs search on the complete structure will act as an insert predicate on the incomplete structure (Why?). Thus, we will not need another argument for the resulting tree.

Code

```
% inserts the element or verifies if the element is already in the tree
insert_it(Key, t(Key, _, _)):-!.
insert_it(Key, t(K, L, _)):-Key<K, !, insert_it(Key, L).
insert_it(Key, t(_, _, R)):-insert_it(Key, R).
```

Follow the execution of:

Query

```
?- trace, T=t(7, t(5, t(3,_,_), t(6,_,_)), t(11,_,_)), insert_it(6, T).
```

```
?- trace, T=t(7, t(5, t(3,_,_), _), t(11,_,_)), insert_it(9, T).
```

### 1.3.3 Predicatul „delete”

Deleting an element from an incomplete binary search tree is very similar with the deletion from a complete binary search tree. Just as in the case of the deletion from incomplete lists, we will keep the same uninitialized variables from the input tree in the output tree.

Code

```
delete_it(_, T, T):- var(T), !.  % element not in tree
delete_it(Key, t(Key, L, R), L):- var(R), !.
delete_it(Key, t(Key, L, R), R):- var(L), !.
delete_it(Key, t(Key, L, R), t(Pred,NL,R)):- !, get_pred(L,Pred,NL).
delete_it(Key, t(K,L,R), t(K,NL,R)):- Key<K, !, delete_it(Key,L,NL).
delete_it(Key, t(K,L,R), t(K,L,NR)):- delete_it(Key,R,NR).

% searches for the predecessor node
get_pred(t(Pred, L, R), Pred, L):- var(R), !.
get_pred(t(Key, L, R), Pred, t(Key, L, NR)):- get_pred(R, Pred, NR).
```

Follow the execution of:

Query

```
?- trace, T=t(7, t(5, t(3,_,_), t(6,_,_)), t(11,_,_)), delete_it(6, T, R).
?- trace, T=t(7, t(5, t(3,_,_), _), t(11,_,_)), delete_it(9, T, R).
```

# 2 Exerciții

Before beginning the exercises, add the following facts which define trees (complete and incomplete binary) into the Prolog script:

Code

```
% Trees:
incomplete_tree(t(7, t(5, t(3, _, _), t(6, _, _)), t(11, _, _))).
complete_tree(t(7, t(5, t(3, nil, nil), t(6, nil, nil)), t(11, nil, nil))).
```

Write a predicate which:

1. Transforms an incomplete list into a complete list *and vice versa*.

Query

```
?- convertIL2CL([1,2,3|_], R).
R = [1, 2, 3].

?- convertCL2IL([1,2,3], R).
R = [1, 2, 3|_].
```

2. Appends two incomplete lists (the result should be an incomplete list – *How many arguments do we need, could we lose one?*).

Query

```
?- append_il([1,2|_], [3,4|_], R).
R = [1, 2, 3, 4|_].
```

3. Reverses an incomplete list (the result should be an incomplete list also). Implement in both types of recursion.

Query

```
?- reverse_il_fwd([1,2,3|_], R).
R = [3, 2, 1|_].

?- reverse_il_bwd([1,2,3|_], R).
R = [3, 2, 1|_].
```

4. Flattens a deep incomplete list (the result should be a simple incomplete list).

Query

```
?- flat_il([[1|_], 2, [3, [4, 5|_]|_]|_], R).
R = [1, 2, 3, 4, 5|_] ;
false.
```

5. Transforms an incomplet tree into a complete tree and vice versa.

| Query |
| --- |
| ?- incomplete_tree(T), convertIT2CT(T, R).<br>R = t(7,t(5,t(3,nil,nil),t(6,nil,nil)),t(11,nil,nil))<br><br>?- complete_tree(T), convertCT2IT(T, R).<br>R = t(7, t(5, t(3, _, _), t(6, _, _)), t(11, _, _)) |

6. Performs a preorder traversal on an incomplete tree, and collects the keys in an incomplete list.

| Query |
| --- |
| ?- incomplete_tree(T), preorder_it(T, R).<br>R = [7, 5, 3, 6, 11\|_] |

7. Computes the height of an incomplete binary tree.

| Query |
| --- |
| ?- incomplete_tree(T), height_it(T, R).<br>R=3 |

8. Computes the diameter of a binary incomplete tree.

$$diam(T) = \max\{diam(T.left), diam(T.right), height(T.left) + height(T.right) + 1\}$$

| Query |
| --- |
| ?- incomplete_tree(T), diam_it(T, R).<br>R=4 |

9. Determines if an incomplete list is a sub-list in another incomplete list.

| Query |
| --- |
| ?- subl_il([1, 1, 2\|_], [1, 2, 3, 1, 1, 3, 1, ***1, 1, 2,*** 3,4\|_]).<br>true<br><br>?- subl_il([1, 1, 2\|_], [1, 2, 3, 1, 1, 3, 1, 1, 1, 3, 2, 4\|_]).<br>false |

10. Write the append_il/2 predicate that concatenates two incomplete lists using only two arguments (without an argument for the result).