9 Incomplete structures – lists and trees

Incomplete structures are special Prolog data elements, with the following particularity: instead of having a constant element at the end – such as [] for lists or nil for trees – they end in a free variable.

Example 9.1: Below you have a few examples of incomplete structures:

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

Incomplete structures, or 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 (addition 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!*) and they 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.

9.1 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. *Examples 9.1.a* and *9.1.b* show instances of such structures.

Incomplete lists are traversed in the same way as complete lists, using the [H|T] pattern; the difference comes when we have to process the end of the list – we are no longer dealing with [] at the end of the list, but with a free variable. This has the following implications on the predicates on incomplete lists:

– testing for the end of the list must ***always*** be performed, even for fail situations – and the fail must be explicit
– when testing for the end of the list, you have to check if you have reached the free variable at the end:
    some_predicate(L, …):-var(L), …
– the clause which checks for the end of the list must ***always*** be the first clause of the predicate (*Why?*)
– 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, in the following we will transform a number of well known list predicates such that they work on incomplete lists.

9.1.1 Member – member_il

Before writing the new predicate, let us check the behavior of the known member predicate (the deterministic version) on incomplete lists.

Exercise 9.1: Trace the execution of the following queries:

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

As you have observed, when the element appears in the list, the predicate member1 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:

```
% must test explicitly for the end of the list, and 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).
```

Exercise 9.2: Trace the execution of the following queries:

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

9.1.2 Insert – insert_il

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:

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

Exercise 9.3: Trace the execution of the following queries:

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

Notice how similar the two predicates are: insert_il and member_il are – the only thing that differs is what to do when reaching the end of the list.

Also, insert_il and member1 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 (*Explain!*).

9.1.3 Delete – delete_il

The predicate for deleting an element from an incomplete list is very similar to its counterpart for complete lists:

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

Again, notice how 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 predicate.

Exercise 9.4: Trace the execution of the following queries:

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

9.2 Incomplete binary search trees

Incomplete trees are another special type of incomplete structures – a branch no longer ends in nil, but in an unbound variable. *Examples 9.1.c* and *9.1.d* show instances of such structures.

The observations in section 9.1 for writing predicates on incomplete lists apply in the case of incomplete trees as well. Thus, in the following we shall apply those observations to develop the same predicates we discussed for lists in the previous section: search_it, insert_it, delete_it – for incomplete binary search trees.

9.2.1 Search – search_it

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:

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

Exercise 9.5: Trace the execution of the following queries:

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

9.2.2 Insert – insert_it

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 here as well the predicate which performs search on the complete structure will act as an insert predicate on the incomplete structure (*Explain!*):

```
insert_it(Key, t(Key, _, _)):-!.
insert_it(Key, t(K, L, R)):-Key<K, !, insert_it(Key, L).
insert_it(Key, t(_, _, R)):- insert_it(Key, R).
```

Exercise 9.6: Trace the execution of the following queries:

1. ?- T = t(7, t(5, t(3, _, _), t(6, _, _)), t(11, _, _)), insert_it(6, T).
2. ?- T = t(7, t(5, t(3, _, _), _), t(11, _, _)), insert_it(9, T).

9.2.3 Delete – delete_it

Deleting an element from an incomplete binary search tree is very similar with the deletion from a complete binary search tree:

```
delete_it(Key, T, T):-var(T), !, write(Key), write(' not in tree\n').
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).

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).
```

Exercise 9.6: Trace the execution of the following queries:

1. ?- T = t(7, t(4, t(3, _, _), t(6, t(5, _, _), _)), t(11, _, _)), delete_it(7, T, R).
2. ?- T = t(7, t(4, t(3, _, _), t(6, t(5, _, _), _)), t(11, _, _)), delete_it(3, T, R).
3. ?- T = t(7, t(4, t(3, _, _), t(6, t(5, _, _), _)), t(11, _, _)), delete_it(6, T, R).
4. ?- T = t(7, t(5, t(3, _, _), _), t(11, _, _)), delete_it(9, T).
5. ?- T = t(7, t(5, t(3, _, _), _), t(11, _, _)), nl, nl, write('Initial: '), write(T), nl, nl, write('Insert 10: '), insert_it(10, T), write(T), nl, nl, write('Delete 7: '), delete_it(7, T, R), write(R), nl, nl, write('Insert 7: '), insert_it(7, R), write(R), nl, nl.

9.3 Quiz exercises

q9-1. Write a predicate which appends two incomplete lists (the result should be an incomplete list also).

q9-2. Write a predicate which reverses an incomplete list (the result should be an incomplete list also).

q9-3. Write a predicate which transforms an incomplete list into a complete list.

q9-4. Write a predicate which performs a preorder traversal on an incomplete tree, and collects the keys in an incomplete list.

q9-5. Write a predicate which computes the height of an incomplete binary tree.

q9-6. Write a predicate which transforms an incomplete tree into a complete tree.

9.4 Problems

p9-1. Write a predicate which takes as input a deep incomplete list (i.e. any list, at any level, ends in a variable). Write a predicate which flattens such a structure.

```
?- flat_il([[1|_], 2, [3, [4, 5|_]|_]|_], R).
R = [1, 2, 3, 4, 5|_] ? ;
no
```

p9-2. (**) Write a predicate which computes the diameter of a binary incomplete tree:
diam(Root) = max{diam(Left), diam(Right), height(Left) + height(Right) + 2}).

p9-3. (**) Write a predicate which determines if an incomplete list is sub-list in another incomplete list.

?- subl_il([**1, 1, 2**|_], [**1**, 2, 3, **1, 1**, 3, **1, 1, 1, 2|**_]).
yes
?- subl_il([**1, 1, 2**|_], [**1**, 2, 3, **1, 1**, 3, **1, 1, 1,** 3, **2|**_]).
no