Difference Lists

1 Theoretical Considerations

In this lesson, we will present you with a new type of list representation. The incomplete lists were an intermediary step towards difference lists. If in incomplete lists, the variable at the end was anonymous, in the case of difference lists this variable is given as a new argument.

An example of how difference lists can make our lives easier: if for complete/incomplete lists to add an element at the end of a list we needed to traverse the whole list, with Difference Lists we can add the element directly to the end of the list (through the use of this new argument).

1.1 Representation

Difference Lists are represented through two parts: the **start** and **end** of the list. For example:

The list L=[1,2,3] can be represented in the form of a difference list by the variables:

S = [1,2,3|X] and E=X, where S represents the start and E the end.

The name of „difference" comes from the fact that the list L can be computed through the difference between S and E.

The empty list, in this case, can be represented by 2 equal variables (S=E). As such, the stopping condition becomes:

- For complete lists: `pred(L,...):- L=[], ... .`
- For incomplete lists: `pred(L,...):- var(L), !, ... .`
- For difference lists: `pred(S, E, ...):- S=E, var(E), !, ... .`

Example:

S: [1,2,3,4]

E: [3,4]

S-E: [1,2]

S-E (the difference list) represents the list obtained by removing part E from S

There are no advantages when using difference lists like in the previous example, but when combined with the concepts of free variables and unification, difference lists become a powerful tool. For example, list [1,2] can be represented by the difference list [1,2|X]-X, where X is a free variable.

1.2 The „add" predicate

A prolog list is accessed through its head and its tail. The setback of this way of viewing the list is that when we have to access the nth element, we must access all the elements before it

first. If, for example, we need to add an element at the end of the list, we must go through all the elements in the list to reach that element.

Code

```
add_cl(X, [H|T], [H|R]):- add_cl(X, T, R).
add_cl(X, [], [X]).
```

The alternative is to use difference lists (represented by two parts, start of the list S and end of the list E):

Code

```
add_dl(X, LS, LE, RS, RE):- RS = LS, LE = [X|RE].
% the LE variable will contain at the first position the added element
```

If we test it in Prolog by asking the following query:

Query

```
?- trace, LS=[1,2,3,4|LE], add_dl(5,LS,LE,RS,RE).
LE = [5|RE],
LS = [1,2,3,4,5|RE],
RS = [1,2,3,4,5|RE]
```

We can follow this in a step-by-step manner:

- Initially we have → LS = [1,2,3,4|LE]
- First operation RS=LS → RS = [1,2,3,4|LE]
- Second operation LE=[X|RE] → RS = [1,2,3,4|[X|RE]]
- Substitute X for its numeric value → RS = [1,2,3,4|[5|RE]]
- Simplify the RS list → RS = [1,2,3,4,5|RE]

To better understand the way the add predicate works, we can imagine the list is represented by two "pointers", one pointing to the start of the list (LS) and the second one to the end of the list (LE), a variable without an assigned value.

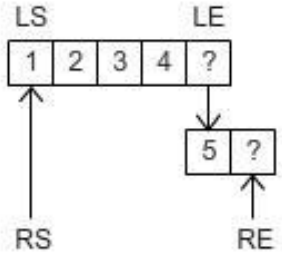

Figure 1. Adding an element at the end of a difference list

The result is also represented by two "pointers". The resulting list will be the input list with the new element inserted at the end. The beginning of the input and result lists is the same so we can unify the result list start variable with the input list start variable (**RS=LS**).

The result list must have an end, just like the input list, in a variable (*RE*), but we must, somehow, modify the input list to add the new element at the end. Because the end of the input

list is a free variable, we can unify it with the list beginning with the new element followed by a new variable, the new end of the list (**LE=[X|RE]**). After the predicate finished execution we can see that the input list *LS* and result list *RS* have the same values, but the end of the input list is no longer a free variable (LE=[5|RE]).

1.3 The „append" predicate

In the case of difference lists, the *append* predicate can be written on one line:

Code

```
append_dl(LS1,LE1, LS2,LE2, RS,RE):- RS=LS1, LE1=LS2, RE=LE2.
```

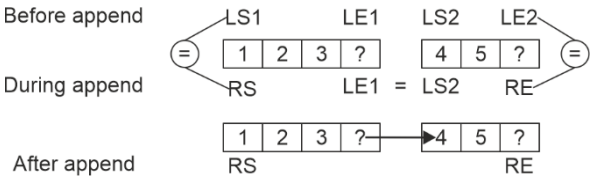

Figure 2. Append two difference lists

If we test it in Prolog by asking the following query:

Query

```
?- trace, LS1=[1,2,3|LE1], LS2=[4,5|LE2], append_dl(LS1, LE1, LS2, LE2, RS, RE).
LE1 = LS2, LS2 = [4, 5|RE],
LE2 = RE,
LS1 = RS, RS = [1, 2, 3, 4, 5|RE].
```

We can follow this in a step-by-step manner:

- Initially we have → LS1=[1,2,3|LE1], LS2=[4,5|LE2]
- First operation RS=LS1 → RS=[1,2,3|LE1]
- Second operation LE1=LS2 → LE1 = [4,5|LE2]
  - We can substitute LE1 in RS → RS=[1,2,3|[4,5|LE2]]
- Third operation RE=LE2
  - We can substitute LE2 in RS → RS=[1,2,3|[4,5|RE]]
- Simplify the RS list -> RS = [1,2,3,4,5|RE]

1.3.1 Quick Sort

Remember the *quicksort* algorithm (and predicate): the input sequence is divided in two parts – the sequence of elements smaller or equal to the pivot and the sequence of elements larger than the pivot; the procedure is called recursively on each partition, and the resulting sorted sequences are appended together with the pivot to generate the sorted sequence:

Code

```
quicksort([H|T], R):-
```

```
    partition(H, T, Sm, Lg),
    quicksort(Sm, SmS),
    quicksort(Lg, LgS),
    append(SmS, [H|LgS], R).
quicksort([], []).
```

Just as for the **inorder** predicate, the quicksort/2 predicate will waste a lot of execution time to append/3 the results of the recursive calls. To avoid this we can apply difference lists again:

Code

```
quicksort_dl([H|T], S, E):- % a new argument was added
    partition(H, T, Sm, Lg), % partition predicate remains the same
    quicksort_dl(Sm, S, [H|L]), %implicit concatenation
    quicksort_dl(Lg, L, E).
quicksort_dl([], L, L). % stopping condition has been changed

partition(P, [X|T], [X|Sm], Lg):- X<P, !, partition(P, T, Sm, Lg).
partition(P, [X|T], Sm, [X|Lg]):- partition(P, T, Sm, Lg).
partition(_, [], [], []).
```

The **partition** predicate remainsthe same, its purpose being to divide the list in two by comparing each element with the pivot. All elements in the list must be accessed for this operation so we cannot improve the performance for the partition predicate.

The **quicksort** predicate works in the same way as before: divides the list in elements larger and smaller than the pivot element and applies quicksort recursively on the each partition. The difference from the original version is the result list, represented by two elements, the start and the end of the list, and, consequently, the way in which the results of the two recursive calls are put together with the pivot (figure below).

Follow the execution of:

Query

```
?- trace, quicksort_dl([4,2,5,1,3], L, []).
?- trace, quicksort_dl([4,2,5,1,3], L, _).
```

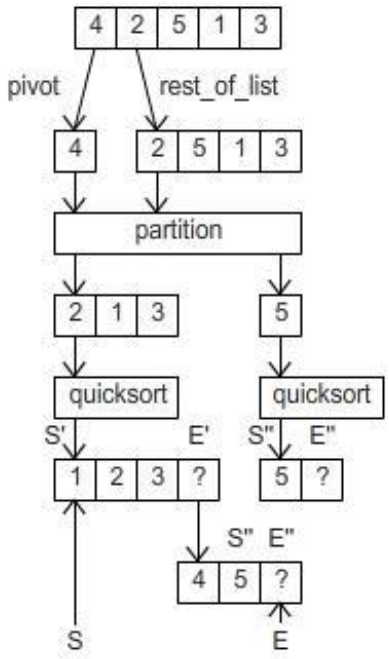

Figure 4. Quicksort with difference lists

1.3.2 Inorder Traversal

The tree traversal predicates are used to extract the elements in a tree to a list in a specific order. The computation intensive part of these predicates is not the traversing itself but the combination of the result lists to obtain the final result. Although hidden from us, prolog will go through the same elements of a list many times to form the result list. We can save a lot of work by changing regular lists to difference lists.

The inorder traversal predicate using regular lists to store the result:

Code

```
inorder(t(K,L,R),List):-
   inorder(L,ListL),
   inorder(R,ListR),
   append1(ListL,[K|ListR],List).
inorder (nil,[]).
```

By executing a query trace on the **inorder/2** predicate we can easily observe the amount of work performed by the append predicate. It is also visible that the append predicate will access the same elements in the result list more than once as the intermediary results are appended to obtain the final result:

Query

```
[. . .]
 8     3 Exit: inorder(t(5,nil,nil),[5]) ?
12     3 Call: append1([2],[4,5],_1594) ?
13     4 Call: append1([],[4,5],_10465) ?
```

```
13     4 Exit: append1([],[4,5],[4,5]) ?
12     3 Exit: append1([2],[4,5],[2,4,5]) ?
[. . .]
22     2 Call: append1([2,4,5],[6,7,9],_440) ?
23     3 Call: append1([4,5],[6,7,9],_20633) ?
24     4 Call: append1([5],[6,7,9],_21109) ?
25     5 Call: append1([],[6,7,9],_21585) ?
25     5 Exit: append1([],[6,7,9],[6,7,9]) ?
24     4 Exit: append1([5],[6,7,9],[5,6,7,9]) ?
23     3 Exit: append1([4,5],[6,7,9],[4,5,6,7,9]) ?
22     2 Exit: append1([2,4,5],[6,7,9],[2,4,5,6,7,9]) ?
[. . .]
```

We can improve the efficiency of the **inorder/2** predicate by rewriting it using difference lists and using the append of difference lists. The **inorder_dl/3** predicate will have 3 parameters: the tree node it is currently processing, the start of the result list and the end of the result list:

Code

```
% when we reached the end of the tree we unify the beginning and end
% of the partial result list – representing an empty list as a difference list
inorder_dl(nil,L,L).
inorder_dl(t(K,L,R),LS,LE):-
   %obtain the start and end of the lists for the left and right subtrees
   inorder_dl(L,LSL,LEL),
   inorder_dl(R,LSR,LER),
   % the start of the result list is the start of the left subtree list
   LS=LSL,
   % key K is inserted between the end of left and the start of right
   LEL=[K|LSR],
   % the end of the result list is the end of the right subtree list
   LE=LER.
```

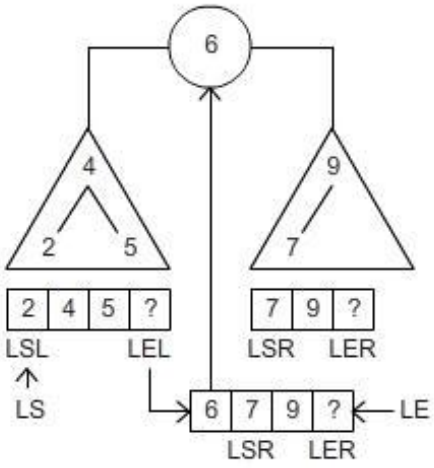

Figure 3. Append two difference lists in the inorder traversal of a tree

Follow the execution of:

Query

```
?- trace, tree1(T), inorder_dl(T,L,[]).
?- trace, tree1(T), inorder_dl(T,L,_).
```

The predicate can be simplified by replacing the explicit unifications with implicit unifications:

Code

```
inorder_dl(nil,L,L).
inorder_dl(t(K,L,R),LS,LE):-
    inorder_dl(L, LS, [K|LT]),
    inorder_dl(R, LT, LE).
```

2 Exercises

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 a complete list into a difference list *and vice versa*.

Query

```
?- convertCL2DL([1,2,3,4], LS, LE).
LS = [1, 2, 3, 4|LE]

?- LS=[1,2,3,4|LE], convertDL2CL(LS,LE,R).
R = [1, 2, 3, 4]
```

2. Transforms an incomplete list into a difference list *and vice versa*.

Query

```
?- convertIL2DL([1,2,3,4|_], LS, LE).
LS = [1, 2, 3, 4|LE]

?- LS=[1,2,3,4|LE], convertDL2IL(LS,LE,R).
R = [1, 2, 3, 4|_]
```

3. Flattens a deep list using difference lists instead of append.

Query

```
?- flat_dl([[1], 2, [3, [4, 5]]], RS, RE).
RS = [1, 2, 3, 4, 5|RE] ;
false
```

4. Traverses a **complete** tree in pre-order and post-order using difference lists in an implicit manner.

Query

```
?- complete_tree(T), preorder_dl(T, S, E).
S = [6, 4, 2, 5, 9, 7|E]

?- complete_tree(T), postorder_dl(T, S, E).
S = [2, 5, 4, 7, 9, 6|E]
```

5. Collects all even keys in a **complete** binary tree, using difference lists.

Query
?- complete_tree(T), even_dl(T, S, E). S = [2, 4, 6|E]

6. Collects, from a **incomplete** binary search tree, all keys between K1 and K2, using difference lists.

Query
?- incomplete_tree(T), between_dl(T, S, E, 3, 7). S = [4, 5, 6|E]

7. Collects, from a **incomplete** binary search tree, all keys at a given depth K using difference lists.

Query
? – incomplete_tree(T), collect_depth_k(T, 2, S, E). S = [4, 9|E].