Difference Lists and Side Effects

1 Objectives

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).

2 Theoretical Considerations

2.1 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.

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.

2.2 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.

```
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):

```
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:

```
?- 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|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.

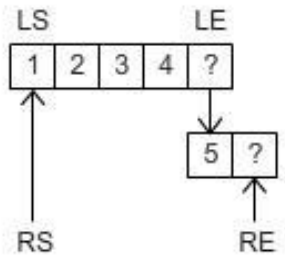

Figure 1. Adding an element at the end of a difference list

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]).

2.3 The „append" predicate

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

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

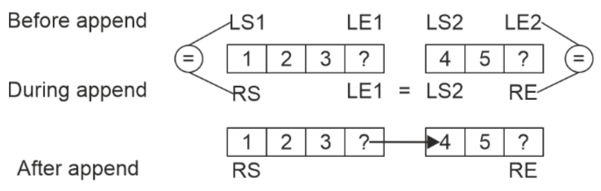

Figure 2. Append two difference lists

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

```
?- 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]

2.3.1 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:

```
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:

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

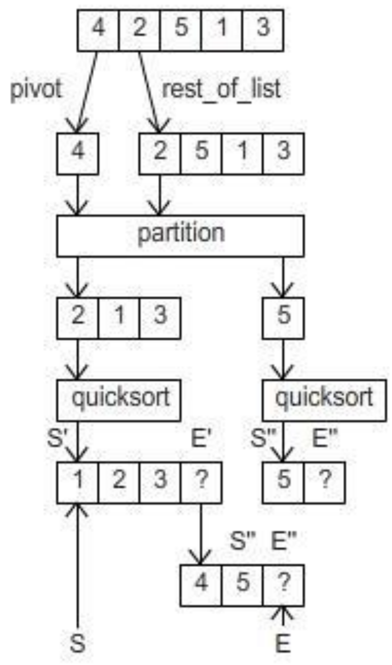

Figure 4. Quicksort with difference lists

2.3.2 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:

```
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:

```
[. . .]
 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:

```
% 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.
```

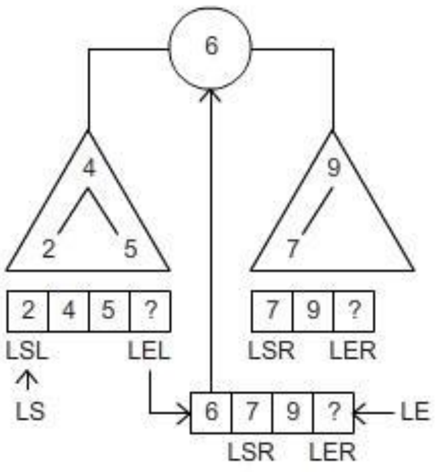

Figure 3. Append two difference lists in the inorder traversal of a tree

Follow the execution of:

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

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

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

2.4 Side Effects

Side effects refer to a series of Prolog predicates which allow the dynamic manipulation of the predicate base:

- *assert*/1 (= *assertz*/1) → adds the predicate clause given as argument as last clause
- *asserta*/1 → adds the predicate clause given as argument as first clause
- *retract*/1 → tries to unify the argument with the first unifying fact or clause in the database. The matching fact or clause is then removed from the database
- *retractall*/1 → deletes all clauses that unify with the argument (in SWI it will always succeed, even if nothing is deleted)

Predicates defined and/or called in the „.pl” file are called static predicates. Predicates which can be manipulated via assert/retract statements at run-time are called dynamic predicates. As opposed to the static predicates that we have seen so far, dynamic predicates should be declared as such.

In Prolog, a predicate is either static or dynamic. A static predicate is one whose facts/rules are predefined at the start of execution, and do not change during execution. Normally, the facts/rules will be in a file of Prolog code which will be loaded during the Prolog session. Sometimes, you might want to add extra facts (or maybe even extra rules) to the predicate, during execution of a Prolog query, using "assert/asserta/assertz", or maybe remove facts/rules using "retract/retractall". To do this, the predicate must be declared as dynamic.

However, when the interpreter sees an assert statement on a new predicate, it implicitly declares it as dynamic. If we want to manipulate a predicate that appears in the „.pl”, then we need to set it as a dynamic predicate by adding the following line to the start of the file:

```
:-dynamic <predicate_name>/<arity>.
```

The important aspects you need to know when working with side effects:

- Their **effect is maintained in the presence of backtracking**, i.e. once a clause has been asserted, it remains on the predicate base until it is explicitly retracted, even if the node corresponding to the *assert* call is deleted from the execution tree (e.g. because of backtracking).
- ***assert* - always succeeds; doesn’t backtrack.**
- ***retract* - may fail and backtrack,** which invalidates temporarily the deletion for predicates within the same body of the clause with the retract call that were called before retract**;** retract respects the *logical update view* - it succeeds for all clauses that match the argument when the predicate was ***called***.

To better understand how it works, try to execute the following query:

```
?-	assert(insect(ant)),
assert(insect(bee)),
retract(insect(A)),
writeln(A),
retract(insect(B)),
fail.
```

You have probably observed that this query will output:

```
ant
bee
true
```

This is because even if the second call to retract will also delete the fact *insect(bee),* when backtracking reaches the first retract call, the clause is still present in its logical view - it doesn't see that the clause has been deleted by the second retract call. As such, *insect(A)* can still unify with „bee".

Prolog also has a **retractall/1** predicate, with the following behavior: it deletes all predicate clauses matching the argument. In some versions of Prolog, retractall may fail if there is nothing to retract. To get around this, you may choose to assert a dummy clause of the right type. In SWI Prolog, however, retractall succeeds even for a call with no matching facts/rules.

Dynamic database manipulation via assert/retract can be used for storing computation results (memorisation/caching), such that they are not destroyed by backtracking. Therefore, if the same question is asked in the future, the answer is retrieved without having to recompute it. This technique is called memorisation, or caching, and in some applications it can greatly increase efficiency. However, side effects can also be used to change the behaviour of predicates at run-time (meta-programming). This generally leads to dirty, difficult to understand code. In the presence of heavy backtracking, it gets even worse. Therefore, this non-declarative feature of Prolog should be used with caution.

An example of memorisation of partial results, using side effects, is the following predicate which computes the nth number in the *fibonacci sequence* (you have already seen the less efficient version in the second lab session):

```
:-dynamic memo_fib/2.

fib(N,F):- memo_fib(N,F), !.
fib(N,F):-
   N>1,
   N1 is N-1,
   N2 is N-2,
   fib(N1,F1),
   fib(N2,F2),
   F is F1+F2,
   assertz(memo_fib(N,F)).
fib(0,1).
fib(1,1).
```

Follow the execution of (run the queries sequentially):

```
?- listing(memo_fib/2). % lists all definitions of the predicate memo_fib with 2 arguments
?- fib(4,F).
?- listing(memo_fib/2).
?- fib(10,F).
?- listing(memo_fib/2).
?- fib(10,F).
```

2.4.1 Printing memorised results

Whenever you want to collect all the answers that you have stored on your predicate base via assert statements, you can use **failure driven loops** which *force Prolog to backtrack* until there are no more possibilities left. The pattern for a failure driven loop which reads all stored clauses - say for predicate *memo_fib/2* above - and prints all the *fibonacci numbers* already computed:

```
print_all:-
   memo_fib(N,F),
   write(N),
   write(' - '),
   write(F),
   nl,
   fail.
```

```
print_all.
```

We will make use of the backtracking technique (by forcing *fail*) to traverse all clauses of the *memo_fib/2* predicate added to the knowledge base through *assert*.

Follow the execution of:

```
?-print_all.
?-retractall(memo_fib(_,_)).
?-print_all.
```

2.4.2 Collecting memorised results

To collect results in a list, we can use the built-in predicate: *findall*.

Follow the execution of:

```
?- findall(X, append(X,_,[1,2,3,4]), List).
?- findall(lists(X,Y), append(X,Y,[1,2,3,4]), List).
?- findall(X, member(X,[1,2,3]), List).
```

Let's now see an example of using *side effects* to get all the possible answers to a query: let's write a predicate which computes all the permutations of a list, and returns them in a separate list. We want the query on the predicate to behave like this:

```
?- all_perm([1,2,3],L).
L=[[1,2,3],[1,3,2],[2,1,3],[2,3,1],[3,1,2],[3,2,1]];
no.
```

Assuming that you already know how to implement the *perm/2* predicate - the predicate which generates a permutation of the input list (see the document on Sorting Methods if not) - the *all_perm/2* predicate specification is:

```
all_perm(L,_):-
    perm(L,L1),
    assertz(p(L1)),
    fail.
all_perm(_,R):- collect_perms(R).

collect_perms([L1|R]):- retract(p(L1)), !, collect_perms(R).
collect_perms([]).
```

```
perm(L, [H|R]):-append(A, [H|T], L), append(A, T, L1), perm(L1, R).
perm([], []).
```

Follow the execution of:

```
?- retractall(p(_)), all_perm([1,2],R).
?- listing(p/1).
?- retractall(p(_)), all_perm([1,2,3],R).
```

Questions:

1. Why do I need a **retractall** call before calling **all_perm/2**?
2. Why do I need a ! after the **retract** call in the first clause of **collect_perms/1**?
3. What kind of recursion is used on **collect_perms/1**? Can you do the collect using the other type of recursion? Which is the order of the permutations in that case?
4. Does **collect_perms/1** destroy the results stored on the predicate base, or does it only read them?

3 Exercises

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

```
% 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*.

```
?- 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*.

```
?- 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.

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

4. Generates a list with all the possible decompositions of a list into 2 lists, without using the built-in predicate *findall*.

```
?- all_decompositions([1,2,3], List).
List=[ [[], [1,2,3]], [[1], [2,3]], [[1,2], [3]], [[1,2,3], []] ] ;
false
```

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

```
?- 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]
```

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

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

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

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

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

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