CAP 5602
Summer, 2011

Lesson 5: Lists

The topics

1. updating a counter
2. the list structure
3. some useful built-in predicates for lists

1. Updating a counter

Let us try to implement the increse the counter operation in prolog. The statement

M = M + 1.

yields M = ** + 1.

because = signifies matching.

The statement

M = 1, M is M + 1.

produces

false

because M = 1 is compared with 2, and the result is false.

So, we can do the following: record the initial count with an assert clause, asserta(count(0)). There is no problem if count occurs in the database because asserta places the fact count(0) at the beginning. When we need to update the count, we use the sequence of statements,

count(N), retract(count(_)), M is N + 1, asserta(count(M)).

The first goal reads, the count, the second erases the first count fact in the database, the 3rd increases the count, and the fourth, writes the new count at the beginning of the database.

Example

The program below counts how many times the predicate alpha is satisfied in the database.

/* count the number of times the alpha(X) predicate is satisfied in the data base */

loop :- asserta(count(0)), count_alpha, print_count.

% counts how many times alpha was satisfied.

```
count_alpha :- alpha(X), update_count, fail.
count_alpha.
% update the count
update_count :- count(N), retract(count(_)), M is N + 1,
    asserta(count(M)),!.            the ! marks the end of the loop body
print_count :- write('The number of alphas is : '), count(N), write(N),
    write('.'), retract(count(_)).          clean up the data base
alpha(a).
alpha(b).
alpha(c).
alpha(X):- a(X), b(X).           the rule is satisfied twice, by e and f
a(d).
a(e).
b(e).
a(f).
b(f).
```

The alpha(X) clause is satisfied 5 times, for X = a,b,c,e,f.
When we enter the querry
?- loop.
prolog answers
The number of alphas is : 5.

2. Lists

Lists are data structures that store sequences of items. An empty list has no items. A non-empty list has a *head* and a **tail**. In prolog the empty list is denoted by the atom []. For the non-empty list we can write the items, separated by commas, between the brackets as in [a,b, 1,2, date(may,27,2011)]. We can find the head and the tail of a non-empty list by unifying it with [Head | Tail].

Head will give the head and Tail the tail of the list. For example,

?- [1,2,3,4] = [Head | Tail].

produces

Head = 1, Tail = [2,3,4].

In prolog lists are structures, that have the operator |, the vertical bar. The list [1,2,3,4] is represented by the structure

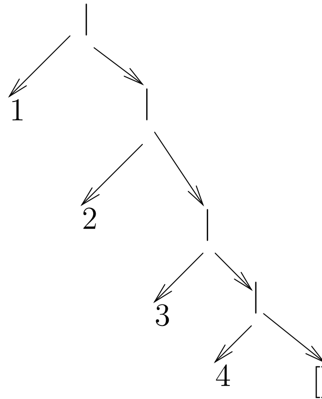

Figure 1: The tree representation of [1,2,3,4]

[1 | [2 | [3 | [4 | []]]]]. The tree repesentation of this list is displayed in Figure 1.

3. Some built-in predicates for lists

We will discuss 7 built-in predicates for lists, member/2, append/3, length/2, reverse/2, delete/3, permutation/2, flatten/2.

1. member(X, List) is satisfied if the item X occurs is List. If X occurs N times, member is satisfied N times.

The predicate can be written as

member(X,[X|_]).

member(X,[_—Tail]) :- member(X, Tail).

Let us see what is happenning when we ask the query

?- member(1, L).

We get the answer

L = [1|_G325]

This tells us that L is a list that has 1 as its head and the tail _G325 is unknown.

If we type a semicolumn and then hit the return key we get

L = [_G324, 1|_G328]

i.e. a list that has 2 as the second item. The first item and the rest of the tail are unknown. If we continue to type ; followed by return we obtain

L = [_G324, _G327, 1|_G331]

and so on. The variables _G325, _G324, etc are system variables.

Let us trace the query to see how we got these results. As we see from

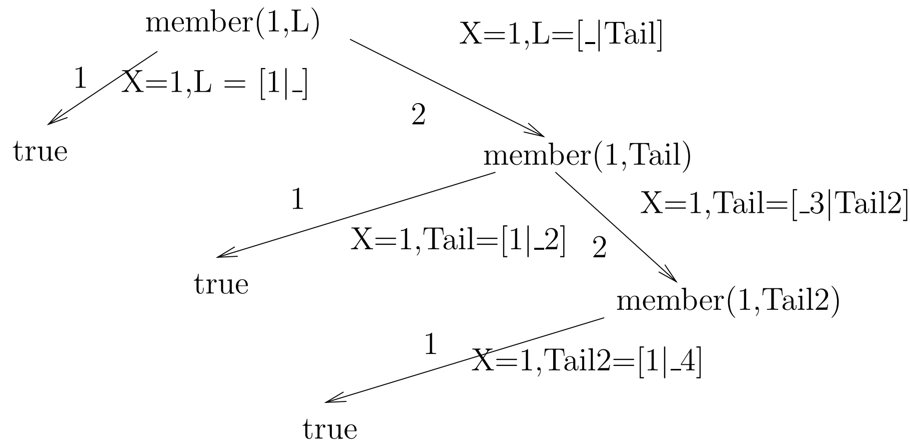

Figure 2: Tracing member(1,L)

Figure 2, the goal member(1,L), can be matched with the first clause of the member predicate.

We get L = [1|_], or L = [1|_G325] for prolog which uses its own system variables.

If we press ; return, the goal is matched with the second clause and we get the new goal member(1,Tail). In turn, this is satisfied by the first clause, and we get Tail = [1|_2]. We replace Tail in the substitution L = [_|Tail] and get L = [_, 1 | _2], or L = [_G324, 1|_G328] in prolog.

Exercise : Rewrite the above predicate such that member is satisfied only once, regardless of the number of occurrences.

2. append(List1,List2,List) is satisfied if List is the concatenation of List1 and List2. The query

?- append([1,2],[a,b],L).

will produce

L = [1, 2, a, b].

This predicate can be written as

append([],L,L).

append([Head | Tail] , List1, [Head — List2]) :- append(Tail, List1, List2).

The predicates member and append were written using the tail recursion, i.e. we process the head and then recursively process the tail.

3. length(List, N) is satisfied if N is the number of items in List.

The query

?- length([a,b,c,d],N).

produces

N = 4.

We can write the length predicate ourselves using tail recursion.

```
length([],0).          % the basis of recursion
% the tail recursion
length([Head | Tail], M):- length(Tail,N), M is N + 1.
```

This predicate is rather inefficient in using the stack; if the list has 10,000 items, it makes 10,000 recursive calls. The program below uses only 2,000 calls.

```
length([],0).
length([X],1).
length([X,Y],2).
length([X,Y,Z],3).
length([X,Y,Z,U],4).
length([X,Y,Z,U,V],5).
% the tail recursion
length([X,Y,Z,U,V | Tail], M):- length(Tail,N), M is N + 5.
```

In the last program we used 6 base clauses instead of 1.

4. reverse(List1, List2) is satisfied if List2 is the reverse of List1.

The query

?- reverse([a,b,c],L).

produces the answer

L = [c,b,a].

We can write the reverse predicate using tail recursion.

```
reverse([],[]).
% reverse the tail and append the head of the reversed tail
reverse([Head | Tail], List) :- reverse(Tail, Reversed_Tail),
    append(Reversed_Tail, [Head], List).
```

5. delete(List1,Item,List2) is satisfied if List2 is obtained by deleting all occurrences of Item from List1.

For example, if we ask the query

?- delete([1,2,1], 1, L).

we get

L = [2].

We can write the clauses of delete using tail recursion.

```
delete([],X,[]).          % the base case
delete([X | Tail1],X, List2):- delete(Tail1,X,List2). % delete the head
delete([Y | Tail1], X, [Y — Tail2]):- delete(Tail1,X,Tail2).
```

6. permutation(List1, List2) is satisfied if List2 is a permutation of List1.

The query

?- permutation([1,2],L).

produces

L = [1, 2] ;

L = [2, 1] .

Here we pressed ; return after the first match.

How can we write the permutation clauses? One way to see how we can generate permutations. If List1 is empty, it has only one permuation, []. If List1 is not empty then we delete the first item of List1, getting a reduced list RList1. We recursively find all permutations of RList1 and insert the deleted item in front of each one. Then we repeat the procedure by deleting the second item of List1, and so on, until the last item.

We get the clauses below.

```
perm([],[]).
perm(List1,[X| RList2]):- del(X,List1,RList1), perm(RList1,RList2).
```

It remains to write the predicate del(X,List1,List2) that deletes one item X from List1 giving List2. Again, we use tail recursion.

```
del(X,[X|Tail1],Tail1). % delete the head
del(X,[Y|Tail1], [Y | Tail2]):- del(X, Tail1, Tail2). % delete from the tail
```

7. flatten(List, ItemList) removes all the brackets that are inside List, giving ItemList, as we can see from the query below.

?- flatten([1,[2,[3],f(a,b),[]]],L).

L = [1, 2, 3, f(a, b)].

This predicate is useful when we want to find the items of a list of lists. How do we write such a predicate? The answer is the program below.

```
% flatten(List1,List2) premoves all the brackets inside List1 giving List2.
flatten([],[]). % the base case
flatten([H|Tail],L):- % H is a list
    flatten(H,L1), % flatten the head
    flatten(Tail,L2), % flatten the tail
```

```
    append(L1,L2,L),!. % concatenate the two lists
flatten([H|Tail],[H|Tail1]):- flatten(Tail,Tail1). % H is not a list
```

We put a cut at the end of clause 2 because we do not want H = [] and H = [A|B] to be processed again by clause 3.