CSEN403 – Concepts of Programming Languages

Topics:

Logic Programming Paradigm: PROLOG

Binary Trees

Lists in Prolog

Originally conducted by :

Prof. Dr. Slim Abdennadher ,

Dr. Nourhan Ehab

Edited by **Dr. Yomna M.I. Hassan**

Feb, 2025

Recap

- Recursion
- Successor Notation

- Reminder: Your quiz is Today, be present at your designated exam hall 10 minutes before the quiz time.
- Preliminary schedule for submissions of the course other than midterm and final is shared on CMS.
- Project 1 document is to be shared after the quiz.
- Piazza has been created to organize your communications and make your questions shareable amongst each other

Structures – Tree

- **Binary trees** can be represented as structures with three arguments:

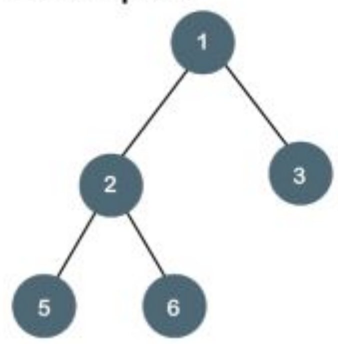
  `bt(Key, LeftSubTree, RightSubTree).`

- The **leaves** of the trees can be denoted by:

  `bt(key, nil, nil)`

  where `nil` is just a constant to represent an empty tree.

- Example:




```
bt( 1,
bt( 2, bt(5,nil,nil), bt(6,nil,nil) ) ,
bt(3,nil,nil) )
```

Example

Implement a predicate `sum_nodes/2`. `sum(T,S)` is true if `S` is the sum of all of the nodes in the tree `T`.

Example

Implement a predicate sum_nodes/2. sum(T,S) is true if S is the sum of all of the nodes in the tree T.

```
sum_nodes(nil, 0).
sum_nodes(bt(Key,L,R), S):-
                    sum_nodes(L, SL),
                    sum_nodes(R, SR),
                    S is Key + SL + SR.
```

Example

Implement a predicate `count_leaves/2`. `count_leaves(T,L)` is true if `L` is the number of leaves in the tree `T`.

Example

Implement a predicate count_leaves/2. count_leaves(T,L) is true if L is the number of leaves in the tree T.

```
count_leaves(nil, 0).
count_leaves(bt(_,nil,nil), 1).
count_leaves(bt(Key, L, R),Leaves) :-
                    count_leaves(L, LL),
                    count_leaves(R, LR),
                    Leaves is LL + LR.
```

List Data Structure

- A useful data structure to be able to list a number of items and go through them.

- Notation: `[H|T]` where
  - `H` is the head of the list (first item); and
  - `T` is the tail of the list (everything without the head).

Example (Lists Representation)

```
?- X = [1,2,3].
   X=[1,2,3].

?- X=[Y,3,s(0)].
   X=[Y,3,s(0)].
```

- Lists of lists can be represented as well.

Example (Lists Representation)

```
?- [H|T]=[1,2,3].
   H=1,
   T=[2,3].

?- [H|T]=[1,[2,abc]].
   H=1,
   T=[[2,abc]].

?- [H1,H2|T]=[1,2,3,4].
   H1=1,
   H2=2,
   H3=[3,4].
```

- `X` is **member** in a list `L` if `X` occurs in `L`
- `X` is a member in a list if
  - it is the **first** element
  - otherwise `X` is in the **rest of the list**.

Example

Define a predicate `mem/2`. `mem(E,L)` is true if `E` is on of the elements inside `L`.

Examples:

```
?- mem(6,[7,2,6]).
   true
?- mem(10,[6,2,9]).
   false
```

Example

Define a predicate `mem/2`. `mem(E,L)` is true if `E` is on of the elements inside `L`.

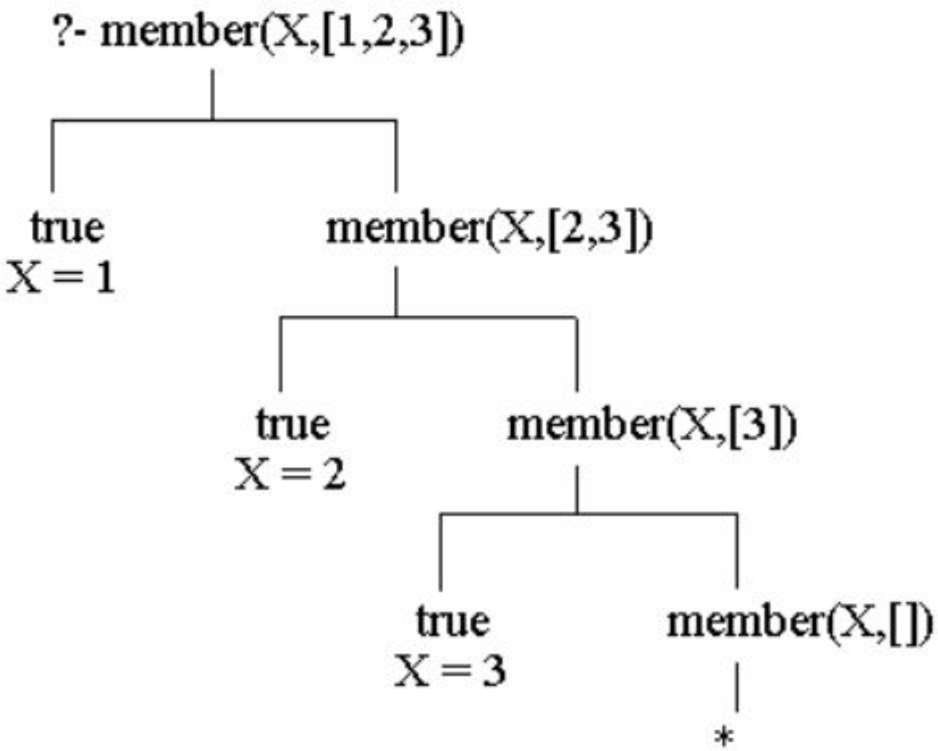
Examples:

```
?- mem(6,[7,2,6]).
   true
?- mem(10,[6,2,9]).
   false
```

```
mem(X,[H|T]):- X=H.
mem(X,[H|T]):- mem(X,T).
```

Search Tree: `member` Example

```
?- member(X,[1,2,3]).
```



Append

Example

Define a predicate app/3. `app(L1,L2,L)` is true if `L` is the result of appending `L2` to `L1`.

Examples:

```
?- app([1,4,2],[9,10],L).
   L=[1,4,2,9,10]
```

Example

Define a predicate app/3. app(L1,L2,L) is true if L is the result of appending L2 to L1.

Examples:

```
?- app([1,4,2],[9,10],L).
   L=[1,4,2,9,10]
```

```
app([],L,L).
app(L1,L2,L):-
              L1=[H|T],
              L=[H|T1],
              app(T,L2,T1).
```

Reverse

Example

Define a predicate `rev/2`. `rev(L1,L)` is true if `L` contains the same elements of `L1` in reversed order.

Examples:

```
?- rev([1,4,2],L).
   L=[2,4,1]
```

Example

Define a predicate `rev/2`. `rev(L1,L)` is true if `L` contains the same elements of `L1` in reversed order.

Examples:

```
?- rev([1,4,2],L).
   L=[2,4,1]
```

Idea: Add an element to the end of `L` using append.

```
rev([],[]).
rev([H|T],L):-
       rev(T,T1),
       app(T1,[H],L).
```

Insert

Example

Define a predicate `insert/3`. `insert(X,L,R)` is true if `X` can be inserted in `L` to produce `R`.

Examples:

```
?- insert(1,[2,3,4],R).
   R = [1, 2, 3, 4] ;
   R = [2, 1, 3, 4] ;
   R = [2, 3, 1, 4] ;
   R = [2, 3, 4, 1] ;
```

Example

Define a predicate `insert/3`. `insert(X,L,R)` is true if `X` can be inserted in `L` to produce `R`.

Examples:

```
?- insert(1,[2,3,4],R).
   R = [1, 2, 3, 4] ;
   R = [2, 1, 3, 4] ;
   R = [2, 3, 1, 4] ;
   R = [2, 3, 4, 1] ;
```

```
insert(X,L,[X|L]).
insert(X,[Y|L],[Y|L1]) :- insert(X,L,L1).
```

Delete

Example

Define a predicate `delete/3`. `delete(X,L,R)` is true if `R` is the result of removing an instance of `X` from `L`.

Examples:

```
?- delete(4,[1,4,2,4],R).
   R=[1,2,4];
   R=[1,4,2]
```

Example

Define a predicate `delete/3`. `delete(X,L,R)` is true if `R` is the result of removing an instance of `X` from `L`.

Examples:

```
?- delete(4,[1,4,2,4],R).
   R=[1,2,4];
   R=[1,4,2]
```

```
delete(X,[X|R],R).
delete(X,[Y|R],[Y|S]) :- delete(X,R,S).
```

```
delete(X,[X|R],R).
delete(X,[Y|R],[Y|S]) :- delete(X,R,S).
```

- When `X` is deleted from `[X|R]`, `R` results.
- When `X` is deleted from the tail of `[Y|R]`, `[Y|S]` results, where `S` is the result of deleting `X` from `R`.
- **Queries:**

```
?- delete(X,[1,2,3],L).
X=1  L=[2,3] ;
X=2  L=[1,3] ;
X=3  L=[1,2] ;
?- delete(3,W,[a,b,c]).
W = [3,a,b,c] ;
W = [a,3,b,c] ;
W = [a,b,3,c] ;
W = [a,b,c,3] ;
```

Insert using Delete

Example

Define a predicate `insert/3`. `insert(X,L,R)` is true if `X` can be inserted in `L` to produce `R`.

Example

Define a predicate `insert/3`. `insert(X,L,R)` is true if `X` can be inserted in `L` to produce `R`.

Idea: `delete(X,L,R)` can be interpreted as "insert `X` into `R` to produce `L`".

```
insert(X,L,R):- delete(X,R,L).
```

Permutation in Lists

```
?- permutation([a, b, c], P).
P=[a, b, c];
P=[a, c, b];
P=[b, a, c];
P=[b, c, a];
P=[c, a, b];
P=[c, b, a]
```

```
permutation ( [], [] ).
permutation ( [ X | L], P) :-
          permutation (L, L1),
          insert (X, L1, P).
```

Accumulator Technique

- **Accumulator**: a data structure to accumulate the subresult during recursive computation
- Often useful to avoid excessive recursive calls.

- At every point, accumulator contains a partial result.
- After all elements are processed, the accumulator contains the final result.

Example in Reverse Function

```
reverse([],X,X).
reverse([X|Y],Z,W) :- reverse(Y,[X|Z],W).

?- reverse([1,2,3],[],A)
          |
   reverse([2,3],[1],A)
          |
   reverse([3],[2,1],A)
          |
   reverse([],[3,2,1],A)
          |
        true

reverse(A,R) :- reverse(A,[],R).
```

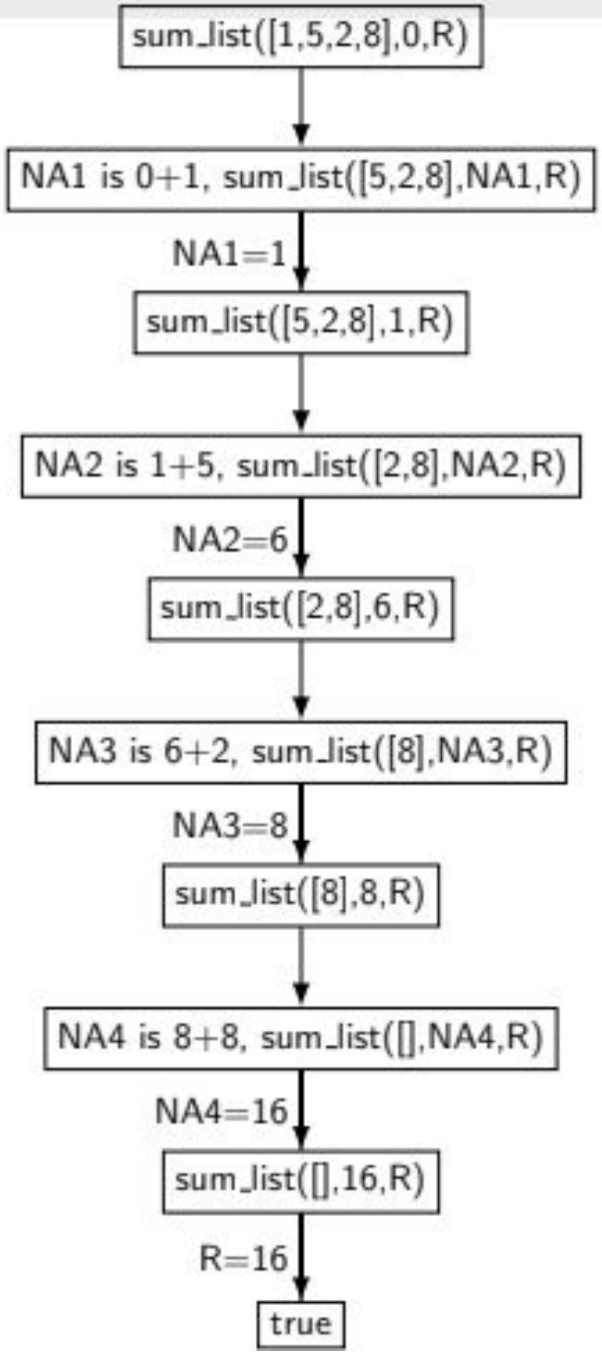
Example on Summation of Values in List

Example

```
sum_list(L,S):- sum_list(L,0,S).
sum_list([], ResultSoFar, FinalResult):-
            FinalResult=ResultSoFar.
sum_list([H|T], ResultSoFar, FinalResult):-
            NewAcc is ResultSoFar + H.
            sum_list(T,NewAcc,FinalResult).
```

List	Result So Far	Final Result
[1,5,2,8]	0	?
[5,2,8]	0+1=1	?
[2,8]	1+5=6	?
[8]	6+2=8	?
[]	8+8=16	16

Trace


sum_list([1,5,2,8],0,R)
NA1 is 0+1, sum_list([5,2,8],NA1,R)
NA1=1
sum_list([5,2,8],1,R)
NA2 is 1+5, sum_list([2,8],NA2,R)
NA2=6
sum_list([2,8],6,R)
NA3 is 6+2, sum_list([8],NA3,R)
NA3=8
sum_list([8],8,R)
NA4 is 8+8, sum_list([],NA4,R)
NA4=16
sum_list([],16,R)
R=16
true

Recap

1. Lists Representation in Prolog.
2. Manipulating Lists.
3. Accumulator technique.

Next Lecture: Putting Everything Together!