# Example Programming Exercises for Prolog

1. Write a Prolog predicate `countBT(Tree, Count)` to count the number of nodes in a binary tree that have two children. Use an accumulator. `Tree` has the structure

   `bt(data, leftTree, rightTree)`.

   The empty tree is represented by an uninstantiated variable. The predicate `var(X)` returns true if `X` is an uninstantiated variable and false otherwise. Use `cut` or `not` to eliminate multiple counts. You must document your solution.

2. Write a predicate, `nth(N, TheList, TheItem),` which is true if `TheItem` is the `N'th` item in `TheList`. Counting begins at one. `nth(1,Alist,Elem)` is true for the first item in the list.

3. Write a predicate, `index(Matrix, [I1,I2,...,In], Elem),` such that Matrix[I1,I2,...,In] = Elem, in a multidimentional matrix . Assume index value 1 is the first item in the corresponding dimension.

4. Write simple Prolog functions such as the following. Take into account lists which are too short.
   -- remove the N'th item from a list.
   -- insert as the N'th item.

5. Write a predicate `diagOf(theMatrix,theDiag)` where theMatrix is a square matrix and theDiag is the diagonal of the matrix. Use an accumulator.

7. Write a predicate `countBT(Tree, Count)` to count the number of nodes in a binary tree. Use an accumulator. `Tree` has the structure `bt(data,leftTree,rightTree)`. The empty tree is represented by an uninstantiated variable. The predicate `var(X)` returns true if X is an uninstantiated variable and false otherwise. Use cut or not to eliminate multiple counts.

8. Assume the prolog predicate `gt(A, B)` is true when A is greater than B. Use this predicate to define the predicate `addLeaf(Tree, X, NewTree)` which is true if `NewTree` is the `Tree` produced by adding the item X in a leaf node. `Tree` and `NewTree` are binary search trees. The empty tree is represented by the atom `nil`.

9. Write a Prolog predicate `oneChildOnly(Tree,Count)` that is true if `Count` is the number of nodes in the binary tree `Tree` that have only one child. Use an accumulator. The empty tree is represented by the atom nil. Use cut or not to eliminate multiple counts.

10. Write a Prolog predicate, `countLists(Alist, Ne, Nl),` using accumulators, that is true when Nl is the number of items that are lists at the top level of Alist and Ne is the number of empty lists. Suggestion: First try to count the lists, or empty lists, then modify by adding the other counter.

11. Give a set of predicates, `binTree` and `treeData`, that define a binary tree where data has the structure of book ownership, `owns(Person,Book)`. Treat `owns` as a compound term not a rule.

12. Define a predicate `memAnywhere(AList,Blist)` that is true if `AList` is a member within `Blist`. Both `Alist` and `Blist` can be any list structure including sublists nested to any depth. This sounds complex because of `Alist` being a complex list structure but try member in the utilities.pro on complex structures and you will find that pattern matching does a lot for you.

13. Define a predicate `memCount(AList,Blist,Count)` that is true if `Alist` occurs `Count` times within `Blist`. Define without using an accumulator. Use "not" as defined in utilities.pro, to make similar cases unique, or else you may get more than one count as an answer.

    Examples:

    ```
    memCount(a,[b,a],N).
    N = 1 ;
    no

    memCount(a,[b,[a,a,[a],c],a],N).
    ```

```
N = 4 ;
no

memCount([a],[b,[a,a,[a],c],a],N).
N = 1 ;
no
```

14. Define memCount as in the previous exercise except this time use an accumulator.

15. Define a predicate `removeNil(Alist,Blist)` that is true if `Blist` is the same as `Alist` with every empty list, [], removed from the top level. This problem is similar to apply-append in Lisp.

    Use "not" in utilities.pro, to make similar cases unique, or else you may get more than one answer from a query.

    Examples:

```
removeNil([a,[],b,[]],TheList).
TheList = [a,b] ;
no

removeNil([[]],TheList).
TheList = [] ;
no

removeNil([a, [[]], b],TheList).
TheList = [a, [[]], b] ;
no
```

15a. Define a predicate `listCount(AList ,Count)` that is true if `Alist` contains `Count` number of elements that are lists. Define using an accumulator. Use "not" as defined in utilities.pro, to make similar cases unique, or else you may get more than one count as an answer.

    Examples:

```
listCount([b,a],N).
N = 0 ;
no

listCount([b,[a,[a],c],a], 1).
N = 1 ;
no

listCount([b,[a,[a],c],a, []],N).
N = 2 ;
no
```

    Define listCount as in the previous exercise except this time without an accumulator.

16. Define a predicate `removeNil(Alist,Blist,N)` that is true if Blist is the same as Alist with every empty list, [], removed from all levels and N is the number of empty lists, [], removed. The base problem, without the counter, is similar to apply-append in Lisp. You may need to use "not" in utilities.pro, to make similar cases unique, or else you may get more than one answer from a query. Define the predicate using an accumulator.

    Examples:

```
  ?- removeNil([a,[],b,[]],TheList,N).
TheList = [a,b] ,
N = 2;
no

  ?- removeNil([[]],TheList,N).
TheList = [] ,
N = 1;
no

  ?- removeNil([a, [[]], [b, c, [d, []]]],TheList,N).
TheList = [a, [], [b, c, [d]]],
N = 2;
no
```

17. The prefix P of a list L can be defined using append as follows.

    ```
    prefix(P,L) :- append(P,_,L).
    ```

    P is a prefix of L if something, including nil, can be suffixed to P to form L.

    Write a definition of prefix in terms of itself (i.e. no other predicates). Try prefix(P,[a,b,c,d]) and other variations.

18. The suffix S of a list L can be defined using append as follows.

    ```
    suffix(S,L) :- append(_,S,L).
    ```

    S is a suffix of L if something, including nil, can be prefixed to S to form L.

    Write a definition of suffix in terms of itself. Try suffix(S,[a,b,c,d]), and other variations.

19. A sublist Sl of a list L can be defined using append as follows.

    ```
    sublist(Sl,L) :- append(_,Sl,Lt) , append(Lt,_,L).
    ```

20. Sl is a sublist of L if something, including nil, can be prefixed to Sl to form the list Lt and something, including nil, can be suffixed to Lt to form L. In other words, Sl is a sublist of L if there exists a prefix P to Sl and a suffix S to Sl such that L= P || Sl || S (|| means concatenation).

    Write a definition of sublist using only sublist and prefix. Do not use the sublist predicate in utilites.pro (that definition does not satisfy the constraints for this problem). Try sublist(Sl,[a,b,c,d]), and other variations.

21. Write a definition of removeContig(A,B) which has the interpretation that the list B is the same as the list A except that instances of a contiguous sequence of identical items are reduced to one instance.

    Examples:
    ```
    removeContig([a,a,b,b,b,c,c,c],[a,b,c])   ==> yes
    removeContig([a,a,b,b,b,c,c,c],[a,b,b,c]  ==> no
    removeContig([a,b,c,a,b,c],[a,b,c])       ==> no
    removeContig([a,b,c,a,b,c],[a,b,c,a,b,c]) ==> yes
    removeContig([[a],[b],[b],[a],[a],[b]], [[a],[b],[a],[b]]) ==>
    yes
    ```

    Draw diagrams of the cases and create rules to handle each case. Hint to access a specific depth into a list; the following predicate is true if the third item on a list is A, otherwise the predicate is false.

    ```
    thirdIs( A , [ _ , _ , A | R] ).
    ```

    First come up with a definition. Then, if multiple answers are possible use cut to remove invalid backtracking choices.

22. Trace the following fibonacci program with the query fib(3,F).

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

23. Document and explain the execution trace of intersection on the query shown in the following. Clearly show the values of all variables at all times, including when they are renamed (use X1, X2 etc. when X is renamed). Do not trace within member. Back substitute the variables until you know what A is.

    Trace `intersection([c,a,d] , [a,b] , A)`

    Intersection(A,B,C) means A intersect B is C. The rules are given on page 148 of the textbook.

    ```
    Rule 1: intersection([], X, []).
    Rule 2: intersection([X | R], Y, [X | Z])
                  :- member(X,Y), ! , intersection(R, Y, Z).
    Rule 3: intersection([X | R], Y, Z)
                  :- intersection(R,Y,Z).
    ```

    For example, the first few steps are the following. Adopt the convention that items to left of the '=' are in the rule, and items to the right are in the goal.

    Try to match Rule 1.
    ```
    [] = [c,a,d]    X = [a,b]     [] = A
    ```
    The match fails on the first argument.

    Try to match Rule 2.
    ```
    [X|R] = [c,a,d]   Y = [a,b]   [X|Z] = A
    ```
    The match succeeds with
    ```
    X = c   R = [a,d]   Y = [a,b]   Z = still a variable, A is
    unknown
    ```

    So Rule 2 is attempted.
    The goal `member(X,Y) = member(c,[a,b])` fails.
    Therefore Rule 2 fails.

    Try to match Rule 3 ... etc.

24. Document and explain the execution trace of union on the following query. Clearly show the values of all variables at all times, including when they are renamed when rules are used multiple times (use X1, X2 etc. when X is renamed). Do not trace within member. Back substitute the variables until you know what A is.

    Trace `union([a,b,c] , [a,d] , A)`

    Union(A,B,C) means A union B is C. The rules are given on page 148 of the textbook.

    ```
    Rule 1: union([], X, X).
    Rule 2: union([X | R], Y,  Z) :- member(X,Y) , ! , union(R, Y,
    Z).
    Rule 3: union([X | R], Y, [X|Z]) :- union(R,Y,Z). </PRE>
    ```

    For example, the first few steps are the following. Adopt the convention that items to left of the '=' are in the rule, and items to the right are in the goal.

    Try to match Rule 1.
    ```
    [] = [a,b,c]    X = [a,d]    X = A
    ```
    The match fails on the first argument.

    Try to match Rule 2.
    ```
    [X|R] = [a,b,c]   Y = [a,d]   Z = A
    ```
    The match succeeds with
    ```
    X = a   R = [b,c]   Y = [a,d]   Z = still a variable, A is
    unknown
    ```

    So Rule 2 is attempted.
    The goal `member(X,Y) = member(a,[a,d])` succeeds.

    Try the query `union([b,c], [a,d], Z).`
    ... and so on

25. The file chat.pro in the class directory on Ariel contains a Prolog solution to the problem of understanding simple English sentence types. Add the necessary rules to the program to understand the following sentence types in addition to the those already programmed. Do not modify or hand in utilites.pro.

(1) ___ is the ___ of ___. Example: Alfred is the brother of Fred.
(2) Is ___ the ___ of ___? Example: Is Alice the mother of Mary?
(3) Who is the ___ of ___? Example: Who is the father of Eliza?

The last case is more complex. The three "blanks" must be different.

(4) X is the ___ of Y if X is the ___ of Z and Z is the ___ of Y.

Example using people named X, Y and Z (they are not variables!).

X is the aunt of Y if X is the sister of Z and Z is the parent of Y.

Example using people named Alice, Mary and Eliza.

Alice is the aunt of Mary if Alice is the sister of Eliza and Eliza is the parent of Mary.

**Example dialogue**

Notice the responses, especially the new response for the "Who" question.

```
| ?- chat.
|: Alfred is the brother of Fred.
Ok
|: Is Alfred the brother of Fred?
Yes

|: Who is the brother of Fred?
Alfred is.

|: Is Fred the brother of Alfred?
No

|: Alice is the aunt of Mary if Alice is the sister   <== should be
         of Eliza and Eliza is the parent of Mary.    <== one line
Ok
|: Who is the aunt of Mary?
! Existence error in sister/2
! procedure user:sister/2 does not exist
! goal:  sister('Alice','Eliza')

| ?- /* sister has not been defined yet. The database still exists so
  we can restart chat. This comment causes the following message. */
! Syntax error
! between lines 53 and 54
! <<here>>

| ?- chat.
|: Alice is the sister of Eliza.
Ok
|: Eliza is the parent of Mary.
Ok
|: Who is the aunt of Mary?
Alice is.

|: Is Alice the aunt of Mary?
Yes

|: Who is the aunt of Tom?
Cannot find one.

|: Fred is the sister of Derek.
```

```
Ok
|: Derek is the parent of Tom.
Ok
|: Who is the aunt of Tom?      <== Notice the rule for aunt is
Cannot find one.                <== specific to Alice, Mary and
Eliza.

|: Stop.
All done.

yes
| ?-
```

26. Create a new rule that is a modification of the last case in the previous exercise to handle the following template. Both rules can exist simultaneously because the statement is different -- the last "the" is changed to an "a".

   ___ is the ___ of ___ if ___ is the ___ of ___ and ___ is a ___ of ___.

   Where the user will use a "variable" name for X, Y, Z. Furthermore, the variables can have any pattern. For example correct input could be any one of the following among others. The blank spaces in the following do not have to be different for this rule (as they must be different for the previous exercise).

   X is the ___ of Y if Y is the ___ of Z and Z is a ___ of Y.
   X is the ___ of Y if X is the ___ of Z and Y is a ___ of Z.
   X is the ___ of X if X is the ___ of X and X is a ___ of X.

   In this case the X,Y,Z locations are to be variables in the asserted clauses and not atoms as in the previous exercise. As a result a general rule can be added to the database and not just a specific instance. Writing the program will require a predicate to map from the input names X,Y,Z,... to variables in creating the clause to assert.

   Example dialogue illustrating the more general nature of the rule than in the previous exercise. Note the phrase is now "a parent" not "the parent" thereby selecting the generalized parsing rule.

```
| ?- chat.
|: Alice is the aunt of Mary if Alice is the sister   <== should be
         of Eliza and Eliza is a parent of Mary.      <== one line
Ok
|: Fred is the sister of Derek.
Ok
|: Derek is the parent of Tom.
Ok
|: Who is the aunt of Tom?      <== Failed previously, succeeds here.
Fred is.                        <== Although rule mentions Alice, Mary
                                <== and Eliza they are replaced with
|: Stop.                        <== variables not stored as constants.
All done.

yes
| ?-
```

27. The file `ariel:/cs/course/3401/tower.pro` contains a definition of a simplified block world which recognizes towers. Create a file that contains the following items.

   27.1 `consult('tower.pro').` -- so you do not have to consult this file when you consult your file. There is no need for me to see or you to print copies of this file to hand in.

   27.2 Try `tower(t(...))` where `t(...)` is some compound term.

```
t(a,ground).
t(d,t(b,t(a,ground))).
t(b,t(d,t(a,ground))).
```

You can use Prolog to create compound terms for towers for you which you can cut and paste for test cases. You can do this for example by querying with appropriate variables and using ';' to create test cases for the other rules.
-- Try `tower(T).`
-- What happens? Can you get a tower with a top block d? Why not? What is the remedy?

27.3 Create a rule `properTower(Topblock,t(...))` that defines a tower such that the top block of Tower is of type top. Add the fact that "d" is a top block to test your rule.

27.4 Create a set of rules to define a tower with height, `measuredTower(Height,t(...))`, such that `Height` is the number of blocks in the tower. You cannot build on tower as it is of the wrong arity.

27.5 Create a rule that defines a good and safe tower, `goodSafeTower(Tb,Height,Min,Max,t(...))`, such that it is a proper tower as in part b and must be between a minimum and maximum height inclusive.

28. Define a predicate swapPrefixSuffix(K,L,R) such that
K is a sublist of L, and
R is the list obtained by appending the suffix of L following an occurence of K in L, with K and with the prefix that precedes that same occurrence of K in L.

For example, if L = P || K || S, where || is list concatenation, then R = S || K || P. Either or both of P and S could be empty, [].

```
?- swapPrefixSuffix([c,d], [a,b,c,d,e], R).
R = [e,c,d,a,b]

?- swapPrefixSuffix([c,e], [a,b,c,d,e], R).
no.

?- swapPrefixSuffix(K, [a,b,c,d,e],[b,c,d,e,a]).
K = [a] ;
K = [] ;
K = [b,c,d,e] ;
no
```

29. Define swapPrefixSuffix using `swapPrefixSuffix(K,L,R,Acc)`, a 4-tuple predicate with an accumulator, Acc, that holds the prefix of L, before the sublist K. SwapPrefixSuffix and append are the only predicates you need. Hint: Draw diagrams of the lists and sublists involved.

A sublist Sl of a list L can be defined using append as follows.

```
sublist(Sl,L) :- append(_,Sl,Lt) , append(Lt,_,L).
```

Sl is a sublist of L if something, including nil, can be prefixed to Sl to form the list Lt and something, including nil, can be suffixed to Lt to form L. In other words, Sl is a sublist of L if there exists a prefix P to Sl and a suffix S to Sl such that L= P || Sl || S.

Explain why K=[] appears in the last example.

30. The file chat.pro in the class directory on Ariel contains a Prolog solution to the problem of understanding simple English sentence structures. Add the necessary rules to chat.pro to be able to understand the sentence structures implied in `ring.pro` up to and including the dies clauses.

In English people's names are in upper case so one has sentences such as the following to establish the facts.

"Fafner is a dragon.",
"Brunnhilde loves Siegfried.",
"The brother of Fasolt is Fafner.".

There are the rules with general sentence types as the following.

"If X is a hero, then X is tenor"-- Since X is a variable the user should be able to use any variable name. Thus, this rule could be entered as "If TheSinger is a hero, then TheSinger is a tenor".

"If TheCharacter is a valkyrie or TheSinger is a tenor, then they are noisy."

"If TheCharacter is a dwarf and they are the son of AnotherCharacter, then AnotherCharacter is a dwarf."

"If TheCharacter is a dwarf, giant or base, then TheCharacter is nogood."

Finally, there are some queries.

"Who does X love?" -- respond with "Siegfried does." if X = Brunnhilde and "No one." if X is Siegfried.

"Does the brother of Fasolt die?" -- respond with "Yes, Fafner does." or "No, Fafner does not die." or "Fasolt has no brother".

Introduce facts and rules to check out all the responses.

"Which nogood characters are the brothers of someone who dies?" -- respond with "There are no such characters." if none exist, or "They are A, B and C .", where A, B, C are the names of the characters (assuming three such characters exist).

"Who dies?" -- respond with a list of lines like the following. Each character should appear only once. "Indirect reason" is for all but basses and heros.
The bass Fafner.
The hero Siegfried.
Indirect reason Mime.

Hand in a script output of the modified program showing the recreation of the ring cycle data base and having typical queries answered.

## Other useful predicates

To backtrack and collect a list of responses Prolog has the following three predicates.

`bagof(Variable, Predicate, Bag)` – Variable is the name of a variable in the Predicate. Prolog queries the database, every answer is put in the Bag and the query continues as if ";" had been entered. The Bag contains items in the order in which Prolog found them. If the Predicate fails then bagof fails. Load `ring.pro` and try `bagof(Who,dies(Who),Bag).`

`setof(Variable, Predicate, Set)` – Variable is the name of a variable in the Predicate. Prolog queries the database, every answer is put in the Set and the query continues as if ";" had been entered. The Set contains items in sorted order with each item occurring once. If the Predicate fails then setof fails. Load `ring.pro` and try `setof(Who,dies(Who),Set).`

`findall(Variable, Predicate, List)` – findAll is similar to bagof for one argument predicates. With multiple arguments findall tries all possible values while bagof expects a fixed value for the other arguments. If the Predicate fails, findall returns the empty list. Load

`ring.pro` and try `findall(Who,dies(Who),List)` and
`findall(X,brother(X,Y),List).`

The following is a definition of findall. It works similarly to a difference list.

```
findall(X, Predicate, Xlist) :-
    call(Predicate),      /* Query the database */
    assertz(item(X)),     /* Save answer in database */
    fail                  /* Force backtracking */
  ; assertz(item(eol)),   /* Tag end of list */
    collect(Xlist).       /* Get solutions from database */

collect(List) :-
    retract(item(X)), ! ,   /* Get the first remaining item */
    (X = eol  , ! , List=[]  /* Handle the end tag */
    ; L = [X|Rest],          /* Not at end, get remaining items */
       collect(Rest)).
```

```
<P><LI>Write a definition of the predicate
<TT>deeper_atoms(Org_list,Result_list)</TT> that is true when
<TT>Result_list</TT> is the same as <TT>Org_list</TT>, except
each atom in
<TT>Org_list</TT> is embedded in a list by itself.  Nil,
<TT>[]</TT>, is not
an atom.  The following are some examples:
<PRE>    deeper_atoms([a] , [[a]])
    deeper_atoms([[]] , [[]])
    deeper_atoms([a, [], b] , [[a], [], [b]])
    deeper_atoms([a, b, [c, [b, d]], [e]] , [[a], [b], [[c],
[[b], [d]]], [[e]]])
</PRE>
```

31. Define the predicate **odd_list(a_list)** where a_list is a list of atoms. The predicate asserts the list contains an odd number of elements. Do NOT USE the length predicate or numbers.

```
?- odd_list([]).
No
?- odd_list([one]).
Yes.
?- odd_list([one, two]).
No.
?- odd_list([one, two, three]).
Yes.
?- odd_list(one).
No.
```

32. Define a predicate **add_up_list( L, K)** that asserts that Land K are lists of integers where every element in K is the sum of all the elements in L up to the same position.

Precondition: L is an instantiated variable

Examples:

```
?- add_up_list( [1], K).
K = [1];
no

?- add_up_list( [1, 2], K).
K=[1,3] ;
no
```

```
?- add_up_list([ 1, 3, 5, 7], K).
K=[1, 4, 9, 16];
no
```

31.