Name:Yash Mahajan Class:SE IT B Roll No.:04 PID:19101

St. Francis Institute of Technology
Borivli (West), Mumbai-400103
Department of Information Technology

Experiment – 11

1. Aim: To implement List comprehensions using Prolog

2. Objective: After performing the experiment, the students will be able to implement List comprehensions using Prolog

3. Lab objective mapped: To understand, formulate and implement declarative programming paradigm through logic programming (PSO2) (PO1)

4. Prerequisite: basic knowledge of constants and variables in prolog

5. Requirements: The following are the requirements – Prolog Compiler

6. Pre-Experiment Theory:
Lists in Prolog refers to an ordered sequence of elements. The other important points related to lists are-

- A single element in a list can be represented as [a]
- An empty list can be represented as []
- The elements of lists are separated by commas. Compound lists are also possible
- [first, second, third] = [A|B] where A = first and B= [second, third]. The unification here
- succeeds. A is bound to the first item in the list, and B to the remaining list.
- [ ] /* this is a special list, it is called the empty list because it contains nothing */

Now let’s consider some comparisons of lists:

- [a,b,c] unifies with [Head|Tail] resulting in Head=a and Tail=[b,c]
- [a] unifies with [H|T] resulting in H=a and T=[]
- [a,b,c] unifies with [a|T] resulting in T=[b,c]
- [a,b,c] doesn't unify with [b|T]
- [] doesn't unify with [H|T]
- [] unifies with []. Two empty lists always match

7. Laboratory Exercise
A. Steps to be implemented
1. Open SWI-Prolog
2. Go to ....file-> new
3. New prolog editor will open up.
4. Write your program (collection of facts and rules)
5. Save the file in the desired as .pl file
6. Follow the steps -> Save buffer, Compile buffer and Make
7. At the prompt, first change to working directory using cd command
8. Load the required file
9. To check the outputs, fire appropriate queries at the prompt

B. Program Code

1. WAP in Prolog to –

a. Find the length of a list

```
len([],0). %length of list is 0 when list is empty

%Recurse until the end of list and after each invocation returns,add 1 to
length
len([_|T],N):-len(T,N1),N is N1 + 1.
```

b. Append another list to the primary list

```
appender([],L,L). %when List1 is empty List3 is equal to List2

% if List1 and List3 are non-empty lists, they both have the same head
appender([H|List1],List2,[H|List3]):-appender(List1,List2,List3).
```

c. Find if the given variable is an element of list

```
elem(X,[X|_]). %X is a element of a list with head X

%X is a element of the list if X is element of tail of the list
elem(X,[_|T]):-elem(X,T),!.
```

2. Define a predicate merge (L, K, M) which, given two ordered lists of integers L and K, returns an ordered list M containing all the elements of L and K.

```
merge(L,[],L).  %When second List2 is empty List3 is equal to List1
merge([],K,K).  %When first List1 is empty List3 is equal to List2

/*If The Head of List1 is lesser than the List2,
Then List1 and List3 have same head*/
merge([HX|TX],[HY|TY],[HX|M]) :- X < Y, merge(TX,[HY|TY],M).
/*If The Head of List1 is greater than equal to List2,
Then List2 and List3 have same head*/
merge([HX|TX],[HY|TY],[HY|M]) :- X >= Y, merge([HX|TX],TY,M).
```

8. Post Experimental Exercise

A. Questions:

1. Define the terms-> (i) Unification (ii) Resolution. Give suitable examples

(i) Unification:-

Given a goal to evaluate, Prolog works through the clauses in the database trying to match the goal with each clause in turn, working from top to bottom until a match is found. If no match is found the goal fails.
Prolog uses a very general form of matching known as unification, which generally involves one or more variables being given values in order to make two call terms identical. This is known as binding the variables to values. For example, the terms dog(X) and dog(fido) can be unified by binding variable X to atom fido, i.e. giving X the value fido. The terms owns(john,fido) and owns(P,Q) can be unified by binding variables P and Q to atoms john and fido, respectively.
There are certain rules for unification. They are-

1. Two atoms unify if and only if they are the same.
   Example -
   unifying atoms fido and fido succeeds
   unifying atoms fido and 'fido' also succeeds, as the surrounding quotes are not considered part of the atom itself
   unifying atoms fido and rover fails.
2. Two compound terms unify if and only if they have the same functor and the same arity (i.e. the predicate is the same) and their arguments can be unified pairwise, working from left to right.
   Example-
   unifying compound terms likes(X,Y) with likes(john,mary) succeeds
   unifying compound terms dog(X) with likes(john,mary) fails
3. Unification of an atom with a compound term always fails.
   Example- unifying fido with likes(john,mary). will always fail
4. Two numbers unify if and only if they are the same, so 7 unifies with 7, but not with 6.9.
5. Two unbound variables, say X and Y always unify, with the two variables bound to each other.
6. An unbound variable and a term that is not a variable always unify, with the variable bound to the term.
   - X and fido unify, with variable X bound to the atom fido
   - X and [a,b,c] unify, with X bound to list [a,b,c]
   - X and mypred(a,b,P,Q,R) unify, with X bound to mypred(a,b,P,Q,R)
7. A bound variable is treated as the value to which it is bound.
8. Two lists unify if and only if they have the same number of elements and their elements can be unified pairwise, working from left to right.
   - [a,b,c] can be unified with [X,Y,c], with X bound to a andY bound to b
   - [a,b,c] cannot be unified with [a,b,d]
   - [a,mypred(X,Y),K] can be unified with [P,Z,third], with variables P,Z and K bound to atom a, compound term mypred(X,Y) and atom third, respectively.
9. All other combinations of terms fail to unify.

(ii) Resolution:-

In simple words resolution is an inference mechanism. Let's say we have clauses
```
m :- b.
```
And
```
t :- p, m, z.
```
So from that we can infer

```
t :- p, b, z.
```

This is called resolution. Resolution is the process (in Prolog) of determining whether a rule can be proved. It concerns the depth-first search process for finding variable assignments that satisfy rules (and the rules they refer to, etc. recursively).

Example-

```
f(a).
f(b).
g(a).
g(b).
h(b).
k(X) :- f(X), g(X), h(X).
```

To prove query k(Y) is true, Its search process can be seen in the below. The process is as follows:

1. Create a temporary variable _G34 (randomly-named) to stand in for Y. This is an implementation detail, so that if some other rule uses Y (a completely different variable), then the variable names don't collide.
2. The goal is to prove k(_G34). To do so, proving f(_G34), g(_G34), h(_G34) will be enough. This is the new goal.
3. To satisfy the first part of the new goal, f(_G34), the knowledge base is searched. There is no rule for f/1 (the predicate with arity one) but there is a fact: f(a). The first matching fact/rule is tried first. Thus, _G34 gets set to a (from unification).
4. Now, g(a), h(a) is the new goal. g(a) is satisfied just fine, because exactly that term is found in the knowledge base (trivial unification).
5. Now, h(a) is the new goal. But, nothing in the knowledge base unifies with h(a) and there is no h/1 rule, so there is a problem.
6. Go to the last decision point. This was when _G34 was set to a. Try to set it to something else. f(b) is in the knowledge base as well, so go with _G34 = b.
7. The new goal is g(b), h(b) which can be satisfied by following the steps given above. We get Y=b as the output.

2. Describe the list operations

The ability to represent data in the form of lists is such a valuable feature of Prolog that several built-in predicates have been provided for it.

1. The member built-in predicate takes two arguments. If the first argument is any term except a variable and the second argument is a list, the member succeeds if the first argument is a member of the list denoted by the second argument (i.e. one of its list elements). The predicate is defined as-
   ```
   member(X,[X|_]).
   member(X,[_|L]):-member(X,L).
   ```
   The two clauses defining member/2 just state that X is a member of any list with head X (i.e. that begins with X) and that X is a member of any list for which it is not the head if it is a member of the tail.
   Example-
   ```
   ?- member(a,[a,b,c]).
   true.
   ?- member(mypred(a,b,c),[q,r,s,mypred(a,b,c),w]).
   true.
   ?- member(x,[]).
   false.
   ?- member(X,[a,b,c]).
   X = a ;
   ```

```
X = b ;
X = c ;
false.
```

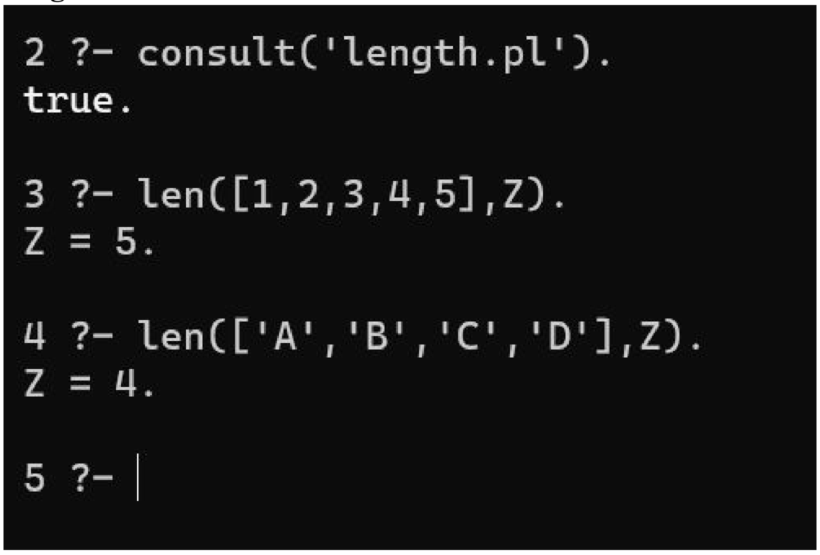
2. The length built-in predicate takes two arguments. The first is a list. If the second is an unbound variable it is bound to the length of the list, i.e. the number of elements it contains.
   It can be defined as-
   ```
   len([],0).
   len([_|T],N):-len(T,N1),N is N1 + 1.
   ```
   Example-
   ```
   ?- length([a,b,c,d],X).
   X = 4
   ?- length([[a,b,c],[d,e,f],[g,h,i]],L).
   L = 3
   ?- length([],L).
   L = 0
   ?- length([a,b,c],3).
   true.
   ?- length([a,b,c],4).
   false.
   ?- N is 3,length([a,b,c],N).
   N = 3
   ```

3. The reverse built-in predicate takes two arguments. If the first is a list and the second is an unbound variable (or vice versa), the variable will be bound to the value of the list with the elements written in reverse order, e.g.
   ```
   ?- reverse([1,2,3,4],L).
   L = [4,3,2,1]
   ?- reverse(L,[1,2,3,4]).
   L = [4,3,2,1]
   ?- reverse([1,2,3,4],[4,3,2,1]).
   true.
   ?- reverse([1,2,3,4],[3,2,1]).
   false.
   ```
   It is defined as -
   ```
   reverse(L1,L2):-rev(L1,[],L2).
   rev([],L,L).
   rev([A|L],L1,L2):-rev(L,[A|L1],L2).
   ```

4. The append built-in predicate takes three arguments. If the first two arguments are lists and the third argument is an unbound variable, the third argument is bound to a list comprising the first two lists concatenated, e.g.
   ```
   ?- append([1,2,3,4],[5,6,7,8,9],L).
   L = [1,2,3,4,5,6,7,8,9]
   ?- append([],[1,2,3],L).
   L = [1,2,3]
   ```
   It is defined as -
   ```
   append([],L,L).
   append([A|L1],L2,[A|L3]):-append(L1,L2,L3).
   ```

B. Results/Observations/Program output:

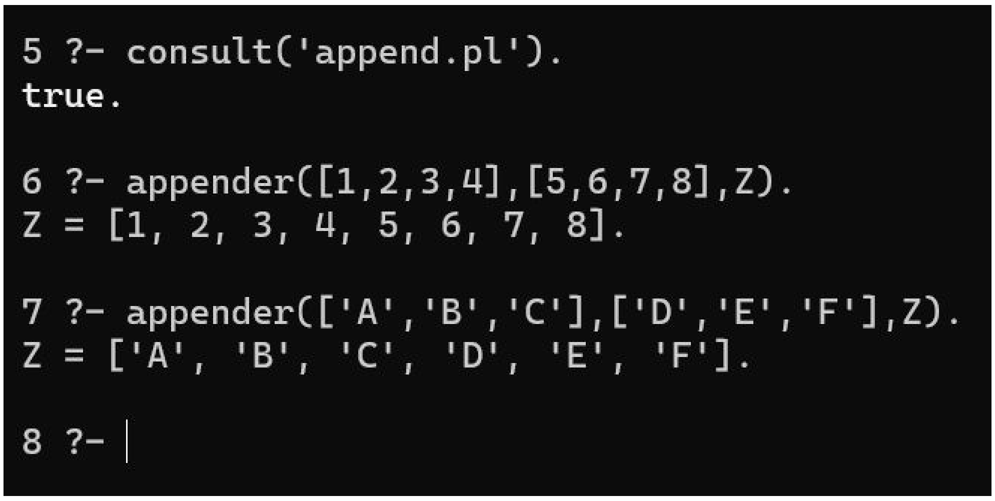
Program 1:-



```
2 ?- consult('length.pl').
true.

3 ?- len([1,2,3,4,5],Z).
Z = 5.

4 ?- len(['A','B','C','D'],Z).
Z = 4.

5 ?- 
```

From this output we can infer that predicate len/2 takes 2 arguments in the form of a list and an unbound variable to return the length of the list. The first clause of the predicate in the code block signifies that length of list is 0 when list is empty, and the second line signifies recursion until the list is empty and then adding 1 to the result of each invocation.

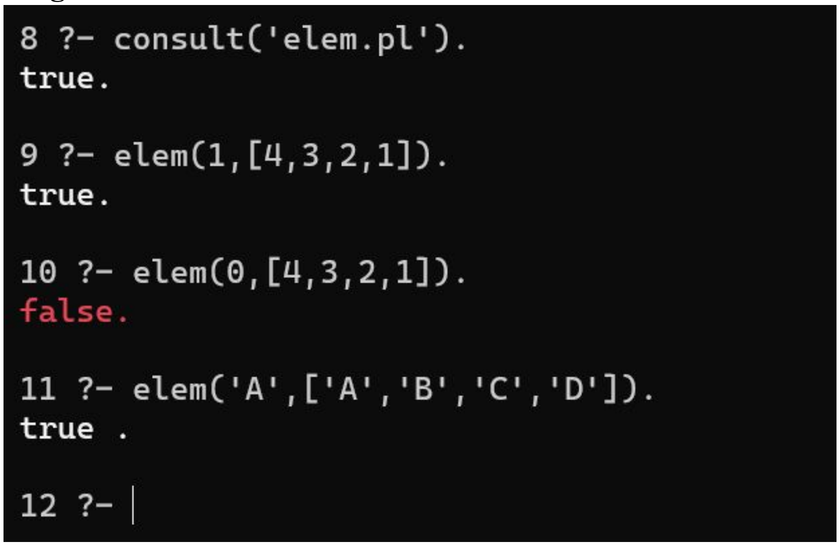
Program 2:-



```
5 ?- consult('append.pl').
true.

6 ?- appender([1,2,3,4],[5,6,7,8],Z).
Z = [1, 2, 3, 4, 5, 6, 7, 8].

7 ?- appender(['A','B','C'],['D','E','F'],Z).
Z = ['A', 'B', 'C', 'D', 'E', 'F'].

8 ?- 
```

From this output we can see that the predicate appender/3 takes 3 arguments in the form of two lists and an unbound variable and returns a list that is resulted by appending 2 lists. The first clause in the code block signifies that if List1 is empty then List3 is bound to List2. The second line signifies that if List1 and List2 are non empty lists, they have the same head.

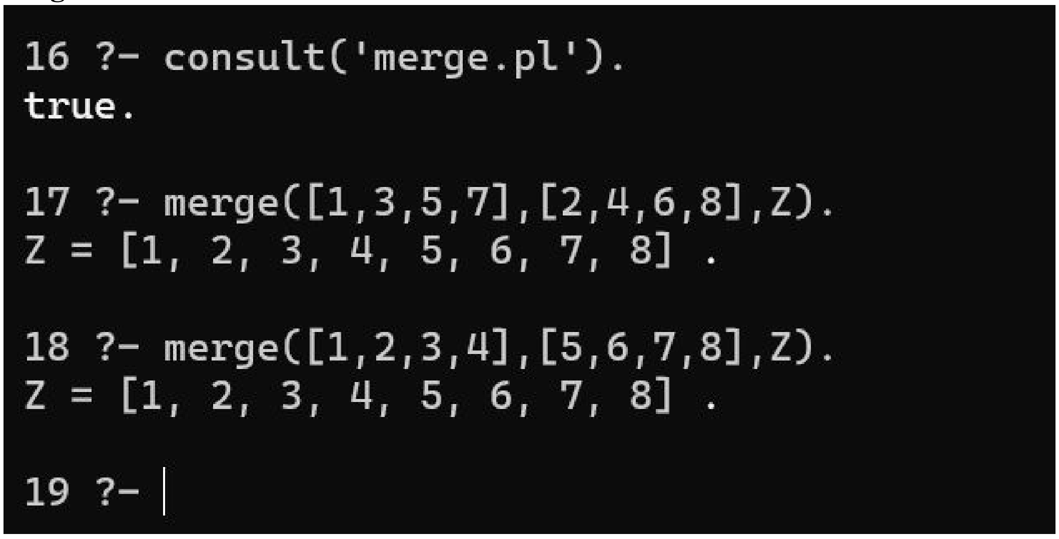
Program 3:-



```
8 ?- consult('elem.pl').
true.

9 ?- elem(1,[4,3,2,1]).
true.

10 ?- elem(0,[4,3,2,1]).
false.

11 ?- elem('A',['A','B','C','D']).
true .

12 ?- 
```

From this output we can see that the predicate elem/2 takes 2 arguments one is any term except an variable and second is a list. The elem predicate succeeds if the first argument is an element of the list. The two clauses defining elem/2 just state that X is an element of any list with head X (i.e. that begins with X) and that X is an element of any list for which it is not the head if it is an element of the tail.

Program 4:-



```
16 ?- consult('merge.pl').
true.

17 ?- merge([1,3,5,7],[2,4,6,8],Z).
Z = [1, 2, 3, 4, 5, 6, 7, 8] .

18 ?- merge([1,2,3,4],[5,6,7,8],Z).
Z = [1, 2, 3, 4, 5, 6, 7, 8] .

19 ?- 
```

From the output we can see that the predicate merge/3 takes 3 arguments the first 2 are lists and the third is an unbound variable to store the result of merging the two lists in ascending order.

C. Conclusion:

In this experiment we have studied the implementation of various list operations that can be performed in Prolog. We have defined predicates to return the length of a list, to append two lists and finding if an element is present in the list. We have also studied the concepts of unification and resolution and how rules in Prolog can be proved.

To perform this experiment we have used the SWI-Prolog 8.2.1 console and editor.

From this experiment we infer that data items can be easily represented in the form of lists in Prolog. Prolog also supports various built-in predicates to manipulate the lists like reverse/2, length/2, append/3 and member/2. We can also define predicates to perform desired operations on lists from left to right using recursion.

D. References:

[1] Michael L Scott, “Programming Language Pragmatics”, Third edition, Elsevier publication
[2] Max Bramer, “Logic Programming with Prolog”, Springer, 2005