Artificial Intelligence Lab
Computer Science CC13P Study Material

Part-II- List Operations

Anupam Pattanayak[1]
Assistant Professor,
Department of Computer Science,
Raja N. L. Khan Women's College (Autonomous),
Midnapore, West Bengal

April 20, 2020

[1]anupam.pk@gmail.com

Contents

1

Prolog Programs - II

We will now see processing of lists. List is a recursive data structure which allows inclusion of diverse objects for knowledge representation. A list is finite sequence of items. We use the *swipl* in Linux.

we will refer the book by Brakto[1], and the book by Bramer[2] for our programs.

1.1 List Basics

We will first see some example of lists, as given below:
$[Ishan]$
$[X, Y]$
$[10, 20, 30, 40]$
$[A, 5, 6, W, Z]$
$[Arun, Barun, Harun, Karun, Tarun]$
$[Subhalakshmi, Padmalakshmi, [Muthulakshhmi, Rangalakshmi], Ishithalakshmi]$
$[\,]$
See the last list - it is empty list. The more interesting is the second last example - recursive list. The very first element in the list is referred as *head.* The subsequent items of the list constitute *tail* (or, *body*) of the list. For the first example of lists above, $[Ishan]$ can be interpreted as $[Ishan, [\,]]$ where $[Ishan]$ is the head, and $[\,]$ is the tail.

With the lists we can do lot of operations such as *insertion*, *deletion*, *unification*, *append*, *search*, and *sort*.

[1]Prolog Programming for Artificial Intelligence by Ivan Bratko
[2]Logic Programming with Prolog by Max Bramer

1.2 Built-in List Predicates

Here, we will see use of some built-in predicates of Prolog for list processing.

1.2.1 Unification

The symbol | is used for unification of lists. It specifies head and tail of the unified list. Run the following commands in swipl.

```

?- X=[Arun,Barun, Karun], Y=[Tarun|X].
```

Observe the output, as shown below.

```
X = [Arun, Barun, Karun],
Y = [Tarun, Arun, Barun, Karun].

?-
```

Take another example. Suppose, we have two lists:
$[Sarasvati, Puri, Bharati, Giri, Tirtha]$ and
$[Sagara, Aranya, Ashrama, Parvata, Vana]$
We want to unify these two lists. So, we run the following in the swipl command prompt.

```
?- X=[Sarasvati, Puri, Bharati, Giri, Tirtha], Y=[Sagara, Aranya
    , Ashrama, Parvata, Vana], W=[X|Y].
```

Observe the output carefully, as shown below.

```
X = [Sarasvati, Puri, Bharati, Giri, Tirtha],
Y = [Sagara, Aranya, Ashrama, Parvata, Vana],
W = [[Sarasvati, Puri, Bharati, Giri, Tirtha], Sagara, Aranya,
    Ashrama, Parvata, Vana].

?-
```

1.2.2 Member

This predicate checks if the first argument is a member of the list specified in second argument. For example, see the following

```
?- member(1,[1,12,25]).
true .
```

Since, 1 is a member of $[1, 12, 25]$ the *member* predicate returns *true*. Note that, we have used · to terminate. If we had used ; to see further output, we get the following:

```
?- member(1,[1,12,25]).
true ;
false.
```

Now, observe the following example:

```
?- member(10,[1,10,2,15,10,25]).
true ;
true ;
false.
```

Since, the item *10* is present more than once, we are getting *true* twice. Note, we have to press ; when the output is awaiting action from our end. Pressing the *Enter* key or · will show just one *true*.

1.2.3 Length

It gives the number of elements a list contains. The built-in predicate *length* accepts two arguments. The first argument is a list and the second argument is an unbound variable that gets bounded to the length of list. we give the following in the *swipl* command prompt to get length of a list.

```
?- length([1,10,2,15,10,25],X).
X = 6.

?-
```

1.2.4 Subset

It checks if a list is a subset of another given list. This built-in predicate *subset* accepts two arguments. The first argument is a list that we want to check if it is a subset of the list specified in second argument. It's usage is illustrated in the following.

```
?- subset([15,25],[10,2,15,10,25]).
true.

?-
```

1.2.5 Append

It appends a list after another. That is it concatenates two lists. This built-in predicate *append* accepts three arguments. The first two arguments are the lists that are to be concatenated. Third argument is an unbound variable that gets bounded to the concaatenated list. It's usage is illustrated in the following.

```
?- append([15,25,35],[10,2,20,30,35],A).
A = [15, 25, 35, 10, 2, 20, 30, 35].

?-
```

1.2.6 Reverse

It reverses a list. The built-in predicate *revers* accepts two arguments. The first argument is a list that we want to reverse. The second argument is an unbound variable that gets bounded to the reverse of the list specified in the first argument. It's usage is shown in the following.

```
?- reverse([15,25,35],A).
A = [35, 25, 15].

?-
```

1.2.7 Sorting

There is a built-in predicate *sort/2*, that is, *sort* predicate with two arguments. First argument is the list. Second one is an unbound variable that keeps the sorted result. It's usage is shown below.

```
?- sort([10, 5, 210, 15, 80, 39, 68],L).
L = [5, 10, 15, 39, 68, 80, 210].

?-
```

1.3 Program to Find Union of Two Lists

We are quite aware of set theoretic union operation. Union of two lists [1,10,12] and [2,12,15,18] is [1,10,12,2,12,15,18]. Following program shows the code for finding union of two lists.

```
% Finds union of two lists
% File name: union.pl

union([],X, X).
union([X|R], Y, Z) :- member(X, Y), !, union(R, Y, Z).
union([X|R], Y, [X|Z]) :- union(R, Y, Z).
```

Now we compile the program *union.pl* and execute it. A sample execution is shown below.

```
?- consult('union.pl').
% union.pl compiled 0.00 sec, 1 clauses
true.

?- union([12,30,45],[15,25,35],A).
A = [12, 30, 45, 15, 25, 35].

?-
```

1.4 Program to Search in a List

We now see the code for searching in a list, as given below.

```
% File name: srch.pl

search(A,[A|_]):- write('Successful Search').
search(A,[_|B]):- search(A,B).
```

Now we compile the program *union.pl* and execute it. Few sample executions are shown below.

```
?- consult('srch.pl').
% srch.pl compiled 0.00 sec, 3 clauses
true.


?- search(25,[15,25,35,45,60]).
Successful Search
true .

?- search(20,[15,25,35,45,60]).
false.

?- search(25,[15,25,35,45,60]).
Successful Search
true .

?- search(25,[15,25,35,45,60,25,65]).
Successful Search
true ;
Successful Search
true ;
false.

?-
```

Observe the last execution. It shows *true* for every time the search result is successful. To prevent this, we can modify the program as shown below. That is, it will display *Search Successful* just once.

```
% File name: search1.pl

search(A,[A|_]):- write('Successful Search '),!.
search(A,[_|B]):- search(A,B).
```

Compile this program and execute it. See the output.

1.5 Program to Check for Palindrome

Following program checks if a given string is palindrome or not. We all know that a string is palindrome if it is same from forward as well as from backward.

```
% Checks if a string is palindrome
% File name: palin1.pl

palindrome(X) :- reverse(X,X).
%reverse([ ],[ ]).
%reverse([X1|Y),R) :- reverse(Y,Y1), conc(Y1,[X1],R).
```

Following shows the compilation and sample executions of it.

```
?- consult('palin1.pl').
% palin1.pl compiled 0.00 sec, 1 clauses
true.

?- palindrome("2002").
true.

?- palindrome("malayalam").
true.

?- palindrome("2020").
false.

?-
```

1.6 Program to Find Maximum

Following program finds maximum from a list.

```
% Finds maximum of a list
% File name: largest.pl

max([X|List],Maxval):- find_max(List,Maxval,X).
find_max([],Cur_max,Cur_max).
find_max([A|L],Maxval,Cur_max):- A>Cur_max, find_max(L,Maxval,A)
    .
find_max([A|L],Maxval,Cur_max):- A=<Cur_max, find_max(L,Maxval,
    Cur_max).
```

Following shows the compilation and sample execution of it.

```
?- consult('largest.pl').
% largest.pl compiled 0.00 sec, 5 clauses
true.

?- max([10, 5, 21, 15, 80, 39, 68],N).
N = 80

?-
```