Lists in Prolog

A list is a collection of items, not necessarily homogeneous. In Prolog, lists are inbuilt data structures. Lists can be used to represent sets, stacks, queues, linked lists, and several complex data structures such as trees, graphs, etc.

Basic Notation and Properties of Lists

- A list in Prolog is an ordered collection of items denoted as `[i1, i2, ..., in]`.
- Unlike arrays in other programming languages, Prolog lists allow direct access only to the first element, called the **Head**. A list can be written as:

  [Head | Rest]

  where `Rest` is the remainder of the list.
- Lists allow heterogeneous data types.
- Nested lists of arbitrary depths are allowed.

Some examples:

```
[]                  % empty list
[a]                 % singleton list
[hello, world]      % 2 element list
[[1,2,3,4], p, this]
[[[1, 2], [3, 4]], [[a, b], [x, y]]]    % nested list
```

Pipe (—) Operator

The pipe operator appends an element at the beginning of a list.

```
[a | L]
```

If `L = [b,c,d]`, then:

$$[a \mid L] = [a, b, c, d]$$

Cut (!)

The `!` (cut) operator is a goal that always succeeds but prevents backtracking.
Example: finding the maximum of two numbers.

```
max_element(X, Y, X) :- X > Y.
max_element(X, Y, Y) :- X =< Y.
```

Query:

```
?- max_element(5, 2, Ans).
Ans = 5
```

To prevent unnecessary backtracking, we use cut:

```
max_element(X, Y, X) :- X > Y, !.
max_element(X, Y, Y).
```

Operations on Lists

1. Find the length of a list

```
list_length([], 0).
list_length([_ | L], N) :-
    list_length(L, N1),
    N is N1 + 1.
```

Query:

```
?- list_length([a, b, c, d], N).
N = 4.
```

2. Membership test

```
is_member(X, [X | _]) :- !.
is_member(X, [_ | Rest]) :-
    is_member(X, Rest).
```

Query:

```
?- is_member(c, [a, b, c, d]).
true.

?- is_member(f, [a, b, c, d]).
false.
```

3. Append two lists

```
append_list([], L2, L2).
append_list([X | L1], L2, [X | L3]) :-
    append_list(L1, L2, L3).
```

Query:

```
?- append_list([a, b, c], [p, q], Ans).
Ans = [a, b, c, p, q].
```

4. Insert at the end

```
insert_end(L, X, NewL) :-
    append_list(L, [X], NewL).
```

Query:

```
?- insert_end([1, 2, 3], 7, Ans).
Ans = [1, 2, 3, 7].
```

5. Delete first occurrence

```
delete_element(_, [], []).
delete_element(X, [X | L], L) :- !.
delete_element(X, [Y | L], [Y | L1]) :-
    delete_element(X, L, L1).
```

Query:

```
?- delete_element(b, [a, b, c, b, d], Ans).
Ans = [a, c, b, d].
```

Built-in Predicates for List Manipulation

Prolog also provides predefined predicates for list operations. Some important ones are:

length/2

The second argument is matched with the length of the list.

```
?- length([elephant, [], [1, 2, 3, 4]], Length).
Length = 3.
```

It can also generate a list of free variables of a given length:

```
?- length(List, 3).
List = [_G248, _G251, _G254].
```

member/2

Succeeds if the element is a member of the list.

```
?- member(dog, [elephant, horse, donkey, dog, monkey]).
true.
```

append/3

Concatenates two lists (works like `append`$_l ist/3$).

last/2

Matches the last element of a list.

```
?- last([a,b,c,d], X).
X = d.
```

reverse/2

Reverses the order of a list.

```
?- reverse([1, 2, 3, 4, 5], X).
X = [5, 4, 3, 2, 1].
```

select/3

Removes the first occurrence of an element from a list.

```
?- select(bird, [mouse, bird, jellyfish, zebra], X).
X = [mouse, jellyfish, zebra].
```

Exercises

Exercise 2.1. Write a Prolog predicate analyse_list/1 that takes a list as its argument and prints out the list's head and tail on the screen. If the given list is empty, the predicate should put out a message reporting this fact. If the argument term isn't a list at all, the predicate should just fail. Examples:

```
?- analyse_list([dog, cat, horse, cow]).
This is the head of your list: dog
This is the tail of your list: [cat, horse, cow]
Yes

?- analyse_list([]).
This is an empty list.
Yes

?- analyse_list(sigmund_freud).
No
```

Exercise 2.2. Write a Prolog predicate membership/2 that works like the built-in predicate member/2 (without using member/2).

Hint: This exercise, like many others, can and should be solved using a recursive approach and the head/tail-pattern for lists.

Exercise 2.3. Implement a Prolog predicate remove_duplicates/2 that removes all duplicate elements from a list given in the first argument and returns the result in the second argument position. Example:

```
?- remove_duplicates([a, b, a, c, d, d], List).
List = [b, a, c, d]
Yes
```

Exercise 2.4. Write a Prolog predicate reverse_list/2 that works like the built-in predicate reverse/2 (without using reverse/2). Example:

```
?- reverse_list([tiger, lion, elephant, monkey], List).
List = [monkey, elephant, lion, tiger]
Yes
```

Exercise 2.5. Consider the following Prolog program:

```
whoami([]).
whoami([_, _ | Rest]) :-
    whoami(Rest).
```

Under what circumstances will a goal of the form whoami(X) succeed?

Exercise 2.6. The objective of this exercise is to implement a predicate for returning the last element of a list in two different ways.

(a) Write a predicate last1/2 that works like the built-in predicate last/2 using recursion and the head/tail-pattern for lists.

(b) Define a similar predicate last2/2 solely in terms of append/3, without explicitly using recursion yourself.

Exercise 2.7. Write a predicate replace/4 to replace all occurrences of a given element (second argument) by another given element (third argument) in a given list (first argument). Example:

```
?- replace([1, 2, 3, 4, 3, 5, 6, 3], 3, x, List).
List = [1, 2, x, 4, x, 5, 6, x]
Yes
```

Exercise 2.8. Prolog lists without duplicates can be interpreted as sets. Write a program that given such a list computes the corresponding power set. Recall that the power set of a set S is the set of all subsets of S. This includes the empty set as well as the set S itself.

Define a predicate power/2 such that, if the first argument is instantiated with a list, the corresponding power set (i.e., a list of lists) is returned in the second position. Example:

```
?- power([a, b, c], P).
P = [[a, b, c], [a, b], [a, c], [a], [b, c], [b], [c], []]
Yes
```

Note: The order of the sub-lists in your result doesn't matter.

Exercise 2.9. Write a predicate longer/2 that takes two lists as arguments and succeeds if the second is longer (has more elements) than the first. Implement your solution using only the tools and techniques introduced so far (in particular, do not make use of any arithmetic expressions, i.e., do not make use of any of the material to be covered in the next chapter). Examples:

```
?- longer([dog,cat,snake], [giraffe,elephant,lion,tiger]).
Yes

?- longer([1,2,3,4,5], []).
No
```

Exercise 2.10. This exercise is about numbers. You are used to representing, say, the number twelve using the decimal system, in which it is written as 12. But you could also use the unary system, in which it can be written as 111111111111. In the next chapter we will see how to work with numbers in the usual decimal system, but you actually already know everything you need to know to work with numbers in the unary system.

Your task will be to implement some basic arithmetical operations for working with unary numbers. We will represent unary numbers as lists of xs of the appropriate length. Thus, five would be [x,x,x,x,x], twelve would be [x,x,x,x,x,x,x,x,x,x,x,x], and zero would be []. In the sequel, all numbers are understood to be such non-negative integers given in unary notation.

(a) The successor of a number is the number we obtain if we add one to it. Thus, for example, the successor of five is six. Write a predicate called successor/2 that will return, in the second argument position, the successor of the number provided in the first argument position. Examples:

```
?- successor([x, x, x], Result).
Result = [x, x, x, x]
Yes

?- successor([], Result).
Result = [x]
Yes
```

(b) Implement a predicate plus/3 to compute the sum of two given numbers. Example:

```
?- plus([x, x], [x, x, x, x], Result).
Result = [x, x, x, x, x, x]
Yes
```

(c) Implement a predicate times/3 to multiply two given numbers. Examples:

```
?- times([x, x], [x, x, x, x], Result).
Result = [x, x, x, x, x, x, x, x]
Yes

?- times([x, x, x], [x, x, x, x, x], Result), write(Result).
[x, x, x, x, x, x, x, x, x, x, x, x, x, x, x]
Result = [x, x, x, x, x, x, x, x|...]
Yes
```

Note: In the last example, the result (a list of fifteen xs) is too long for Prolog to print, so we force printing using the write-command at the end of our query. Make sure your predicate works correctly also when one of the numbers is zero.

Hint: You don't need to use any normal numbers in your program and you should not use any arithmetic operations provided by Prolog (to be covered in the next chapter).