Cairo University
Faculty of Computers & Artificial Intelligence
Department of Computer Science

Artificial Intelligence Course
Lab 3

Lists & Backtracking

Lab objective:

- Using lists in Prolog.
- Controlling backtracking.
- Exercises.

Part 1 – Lists:

- Lists are powerful data structures for holding and manipulating groups of things; a list is simply a collection of terms. The terms can be any prolog data types including structures and other lists.
- Lists are denoted by square brackets with the terms separated by commas.
- For instance, in the list: [ann, X, sport(tennis), [tom,skiing]], the elements are (respectively) an atom, a variable, a structure with functor sport/1, and another list. Notice the nesting of one list within another.
- The empty list is represented by a set of empty brackets [] (without space between them).
- List manipulation can be done by a special notation that allows us to reference to the first element in a list (head) and the list of remaining elements (tail): **[H|T]**. We also need a way of marking the end of a list and in Prolog, this marker is the empty list.
- .(Head, Tail) is the same as [Head|Tail] in prolog. For example:
  - [tennis, tom, skiing] = .(tennis, .(tom, [skiing])).
  - [a,b,c|[d,e]] = [a,b,c,d,e]
  - [a|[]] = [a]
  - [[a,b]|[c]] = [[a,b],c]

Examples:

- **Member -** Define the predicate member that checks if an element is a member of a list.

```
member(X, [X|_]).
member(X, [_|Tail]):-
        member(X, Tail).
```

Now test the predicate in prolog:

?- member(a, [a, b, c]).

true

?- member([c], [a, b|[c]]). % the list is [a, b, c], so c is a member while [c] is not.

false

?- member(X, [a, b, c]).

X = a;

X = b;

X = c.

- **Delete -** Consider programming delete/3, such that delete(X, A, B) is true if X is a member of A and B is the result of removing any one occurrence of X from A.

```
delete(X, [X|Y], Y).
delete(X, [Y|Tail1], [Y|Tail2]):-
        delete(X, Tail1, Tail2).
```

- **BeforeLast -** The following example returns the element before the last element in a list.

```
before_last([X, Y], X).
before_last([H|T], X):-
        before_last(T, X).
```

Exercises: *(these problems may have more than 1 correct answer)*

Problem 1: Use append/3 in order to define suffix/2(Suffixed, List), where suffix/2 checks if the first argument is the suffix of the second argument.

```
%suffix (Suffixed, List), where both arguments are lists

suffix(Suffixed, List):-
      append(_, Suffixed, List).

%append(List1, List2, List3), where List3 is the result of appending List2 to List1
append([], L, L).
append([H|T], L2, [H|NT]:-
      append(T, L2, NT).
```

Problem 2: Use append/3 in order to define prefix/2(Prefixed, List), where prefix/2 checks if the first argument is a prefixed in the second argument.

```
%prefix(Prefixed, List), where both arguments are list

prefix(Prefixed, List):-
      append(Prefixed, _, List).

%append( List1, List2, List3), where List3 is the result of appending List2 to List1
append([], L, L).
append([H|T], L2, [H|NT]):-
      append(T, L2, NT).
```

Problem 3: Use append/3 in order to define last/2(LastElement, List), where last/2 check if the first argument is a last element in the second argument.

```
%last(Last, List), where Last is an element and List is a list

last(Element, List):-
      append(_, [Element], List).

%append( List1, List2, List3), where List3 is the result of appending List2 to List1
append([], L, L).
append([H|T], L2, [H|NT]):-
      append(T, L2, NT).
```

Problem 4: Define the predicate addone/2(L1, L2) with L1 and L2 both lists of integers such that addone is true if for every element of L1 the corresponding element in L2 is obtained by adding 1 to the element of L1. For Example:

```
?- addone([3, -6], L2).
L2 = [4, -5].
```

```
%addone(InList, OutList), where both arguments are lists.

addone([], []).
addone([First| Rest], [Add| Rest1]):-
      Add is First +1,
      addone(Rest, Rest1).
```

Problem 5: Define the predicate adjacent/3(X, Y, Z) that checks if the elements X and Y are adjacent in the list Z. The main idea is to traverse the list Z until we get the two elements X and Y as the head of the list.

```
%adjacent(X, Y, Z), where the first two arguments are elements and the last is a list

adjacent(X, Y, [X,Y|_]).
adjacent(X, Y, [_|T]):-
      adjacent(X, Y, T).
```

Problem 6: Define a predicate that finds the length of a list. The recursive definition of the length of a list is: The length of the list L is just the length of its tail plus one.

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

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

Problem 7: Define a predicate that finds the sum of a list. Use tail recursion to solve this problem.

```
sum(L, R):-
      sum(L, 0, R).

sum([], PR, PR).

sum([X|T], PR, R):-
      S is PR + X,
      sum(T, S, R).
```

Problem 8: Define a predicate that gets the n^{th} element in a list. Use tail recursion. The predicate should also be able to return the index if the element is given.

```
nth(X, L, N):-
      nth(X, L, 0, N).

nth(X, [X|_], I, I).

nth(X, [_|T], I, N):-
     I1 is I+1,
     nth(X, T, I1, N).

?- nth(A,[x,3,5,9,10],4).
A = 10.

?- nth(3,[x,3,5,9,10,3],X). %also works as 'index of'
X = 1;
X = 5;
false.
```

Part 2 – Backtracking:

Consider the following program:

```
hold_party(X):-
     birthday(X),
     happy(X).

birthday(tom).
birthday(fred).
birthday(helen).

happy(mary).
happy(jane).
happy(helen).
```

If we now pose the query:

```
?- hold_party(Who).
```

What happens is:

1. Prolog interpreter tries to bind X to a constant through birthday(X) i.e. birthday(X) as 'subgoal1'.
2. birthday(X) matches with birthday(tom), so X = tom and 'subgoal1' is true. Now it's time to prove happy(tom) as 'subgoal2'.

3. Prolog interpreter traverses the knowledge base to prove happy(tom) but it fails.
4. Prolog interpreter **backtracks** and tries to prove birthday(X) with some value other than tom ('subgoal3').
5. birthday(X) matches with birthday(fred), so X = fred and 'subgoal3' is true. Now it is time to prove happy(fred) as 'subgoal4'.
6. Prolog interpreter traverses the knowledge base to prove happy(fred) but it fails.
7. Prolog interpreter **backtracks** and tries to prove birthday(X) with some value other than tom and fred ('subgoal5').
8. birthday(X) matches with birthday(helen), so X = Helen and 'subgoal5' is true. Now it is time to prove happy(helen) as 'subgoal6'.
9. Prolog interpreter traverses the knowledge base to prove happy(helen) and it succeeds.

Controlling backtracking:

- Controlling backtracking in prolog is done via **'!' operator and it is called the 'cut' operator.** It is inserted anywhere in the rule as another subgoal but it has a special meaning which is:
    - **It is always evaluated as true**; i.e. the subgoal of '!' never fails.
    - If any of the the subgoals before it in the rule is evaluated to **false**, **prolog backtracks** and enters an alternative predicate (i.e. another rule/predicate with the same name and number of parameters).
    - If the subgoals before it in the rule are evaluated to **true**, the values that are associated with variables in these subgoals are confirmed and **no more backtracking will be done on these values** (but it doesn't affect values of variables after the '!' operator, so backtracking will be normal on these variables). This means that if the rule that has the '!' operator has an alternative predicate and the interpreter is at the '!' subgoal, it won't enter the next alternative predicate even if the current predicate (with the '!') fails afterwards.

- *Why control backtracking?*
  - One use of controlling backtracking is **to improve efficiency and that is referred to as a green cut.** For example:

```
pay(X):- gotmoney(X), !.
pay(X):- gotcredit(X), \+ gotmoney(X).
```

    Here the '!' is put to prevent the interpreter from entering the second rule if the first one succeeded. However, in this example, removing '!' operator won't harm as even if the interpreter entered the second rule after entering the first, the second rule won't be true as it has the opposite of the first one. So, here the '!' is put only to prevent interpreter from doing extra useless work (from trying to resolve some rule that is known to be false).

  - In some situations it is required to use the '!' operator **to avoid wrong solutions and that is referred to as a red cut.** For example:

```
factorial(0, 1).
factorial(X, R):-
        X1 is X - 1,
        factorial(X1, R1),
        R is X * R1.
```

    Here the factorial problem is decreased by one in each recursive call until we reach zero which has a known solution, so the query:

```
?- Factorial(5,R).
```

    results in:

    R = 120

    Typing ; for another answer results in:

    ERROR: Out of local stack

    In the first solution, the base case (factorial(0, R)) matched with factorial(0, 1) but entering ';' will make factorial(0, R) match with the rule not the fact and factorial will continue decreasing infinitely. So, to make the base case match with the fact and without backtracing, we need to use '!' to be like:

```
factorial(0, 1):- !.
factorial(X, R):-
        X1 is X - 1,
        factorial(X1, R1),
        R is X * R1.
```

This way the only result is R = 120. This is similar to if-else statements; if the program enters the "then" part, it can't enter the "else" part and the "else" part can't be entered unless the "if" part is false.

Another example is the membership problem (test whether some element is in some list)

```
member(X, [X|L]):- !.
member(X, [_|L]):- member(X, L).
```

Here, without the cut, it would respond by true for each ';' pressed as long as there is more of the element in the list but with the cut, only one answer is given.

- Another way **to control backtracking using the is "fail" keyword** but unlike the '!' which means not to backtrack, "fails" means to break the branch and backtrack (i.e. force backtracking). It is used when the condition of failure is clear while the condition of rejection isn't. For example:

`a(X):- fail, X is 1.` `a(X):- X is 2.`	Executing the query a(N) will fail in the first rule because of "fail" keyword and it will backtrack to the second rule and succeed resulting in N = 2.
`a(X):- !, X is 1.` `a(X):- X is 2.`	Executing the query a(N) will succeed in the first rule because '!' is always true and it will continue by setting X to 1 and the second rule will never be entered.
`a(X):- fail, !, X is 1.` `a(X):- X is 2.`	Executing the query a(N) will fail in the first rule before reaching '!', so

	it will backtrack and enter the second rule (N = 2).
`a(X):- !, fail, X is 1.` `a(X):- X is 2.`	Executing the query a(N) will succeed in the first subgoal in the first rule because '!' is true and it will continue but fail in the second subgoal. However, it won't be able to backtrack and enter the second rule, so the result will be **false**.

Exercises: (these problems may have more than 1 correct answer)

Problem 1: Check whether two numbers are divisible by each other or not using ! and fail.

```
not_divisible(X, Y):-
      0 is X mod Y, !,
      fail.

not_divisible(_, _).
```

Problem 2: Remove duplicated elements from a list to get a list of unique elements.

```
remove_duplicates([], []).

remove_duplicates([Head|Tail], Result):-
       member(Head, Tail),!,
       remove_duplicates(Tail, Result).

remove_duplicates([Head|Tail],[Head|Result]):-
       remove_duplicates(Tail, Result).
```

Here, without the cut, it would respond by wrong answers when ';' is pressed for other results although the first result is the only correct result. However, with the cut, only one answer is given and it is the right one.

Problem 3: Add an element to a list if it is not already in the list.

```
Add(E, L, L):-
       member(E, L), !.

Add(E, L, [E|L]).
```

Problem 4: Partition a list into two lists; one with elements smaller than some value and one with the elements greater than or equal to that value without ability to give wrong results when pressing ';'.

```
partition(_, [], [], []):- !.

partition(P, [H|T], [H|T2], L3):-
       H < P, !, partition(P, T, T2, L3).

partition(P,[H|T],L2,[H|T3]):-
       partition(P, T, L2, T3).
```

Here, without the second cut, it would respond by wrong answers when ';' is although the first result is the only correct result. However, with the second cut, only one answer is given, and it is the right one but the interpreter waits for user to decide whether he wants another result and if he pressed ';' false is returned (i.e. no more results). To remove this useless waiting, we use the first cut.