INST0072: Exercises for Lecture 6

Exercise 6-1

The inbuilt SWI Prolog predicate '`member/2`' behaves identically to the predicate '`element/2`' in the program 'elementAndJoin.pl' [view | download]. The inbuilt SWI Prolog predicate '`append/3`' behaves identically to the predicate '`member/3`'. Predict then test the answers to the following queries:

```
?- X = [p|[q|[r|[]]]].
?- member(X,[p,q,r]).
?- member(X,[p,[q,r]]).
?- member(p,X).
?- member(p,[X|Y]).
?- append([p,q],[r,s],X).
?- append([p,q],X,[p,q,r,s]).
?- append(X,[r,s],[p,q,r,s]).
?- append(X,Y,[p,q,r,s]).
?- append(X,[r,s],Y).
?- append([p,q],X,Y).
?- append(X,Y,Z).
?- append([p,q],[r,s],X),member(Y,X).
```

[Answer]

Exercise 6-2

Run the following query with the program 'elementAndJoin.pl' [view | download] and then draw its search tree:

```
?- join([a, b], [c, d], N).
```

[Answer]

Exercise 6-3

Run the following query with the program 'elementAndJoin.pl' [view | download] and then draw its search tree:

```
?- join(N, M, [a, b]).
```

[Answer]

Exercise 6-4

The built-in infix predicate '`is/2`' is used in Prolog for arithmetic equality. (The right-hand argument of '`is/2`' has to be fully instantiated before the goal is called.) Predict the output of the following queries, then test them at the SWI Prolog prompt:

```
?- X is 6 + 1.
?- X is 6, Y is X + 4.
?- X is 6, X is Y - 4.
```

Write a recursive definition for a predicate 'count_elements/2' which has a list as the first argument and a number as the second argument. The query '`?- count_elements(`*`List`*`, `*`Number`*`)`' should succeed if and only if *`Number`* is the number of elements in *`List`*. The base case for this predicate will be

```
count_elements([], 0).
```

Your answer should reproduce the following example input/output:

```
?- count_elements([a, b, c], X).
X = 3.

?- count_elements([a, b], X).
X = 2.

?- count_elements([a, b, c, d, e, f], X).
X = 6.
```

Finally, test what happens with your answer and the following query:

```
?- count_elements(X, 3).
```

[View Example Solution | Download Example Solution]

Exercise 6-5

Write a recursive definition for a predicate 'odd_or_even/2' which has a list as the first argument and either the constant 'even' or the constant 'odd' as the second argument. The query '?- odd_or_even(*List*, even)' should succeed if and only if *List* contains an even number of elements, and the query '?- odd_or_even(*List*, odd)' should succeed if and only if *List* contains an odd number of elements. The base case for this predicate will be

```
odd_or_even([], even).
```

Your answer should reproduce the following example input/output:

```
?- odd_or_even([a, b, c], X).
X = odd ;
false.

?- odd_or_even([a, b], X).
X = even ;
false.

?- odd_or_even([a, b, c, d, e, f], X).
X = even ;
false.
```

Finally, test what happens with your answer and the following query:

```
?- odd_or_even(X, even).
```

[View Example Solution | Download Example Solution]

Exercise 6-6 [quite challenging!]

Write a recursive definition for a predicate 'reverse/2' which is true if the two arguments are lists, and the second is the reverse of the first. The base case for this predicate will be

```
reverse([], []).
```

You may need to use the built in predicate 'append/3' (or 'join/3') in your definition of the recursive clause. Your answer should reproduce the following example input/output:

```
?- reverse([a, b, c], R).
R = [c, b, a].

?- reverse(L, [a, b, c]).
L = [c, b, a] .
```

[View Example Solution | Download Example Solution | Search Tree for Example Query]

Exercise 6-7 [challenging!]

Write a recursive definition for a predicate 'without_even_elements/2' which is true if the two arguments are lists, and the second list is the same as the first list but with the evenly positioned elements removed. You may want to use the predicates 'reverse/2' and 'odd_or_even/2' from exercises 3-4 and 3-5 in your program. Your answer should reproduce the following example input/output:

```
?- without_even_elements([1,2,3,4],X).
X = [1, 3] ;
false.

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

Finally, test what happens with your answer and the following query:

```
?- without_even_elements(L, [1,3,5]).
```

[View Example Solution | Download Example Solution]

Exercise 6-8

Change the Monkey and Banana program 'monkeyAndBanana.pl' [view | download] so that it does not generate plans with variables (i.e. random positions) in them. To do this, you will have to define a new predicate 'place/1' which describes all possible positions in the room. This will just be three facts in the following order:

```
place(atWindow).
place(middle).
place(atDoor).
```

Add conditions (i.e. sub-goals in the body) to the last three clauses for 'can_do', in terms of 'place'. In the case of the 'walk' and 'push' actions, you also need to add the sub-goal '`Position_before \== Position_after`' to their clauses. '`\==`' is an SWI Prolog infix operator which means not-identical-to, so that the monkey will not be able to travel to a place if it is already there. This sub-goal should be added last in both clauses (why?). Query the new program in SWI Prolog to check it works.

[View Example Solution | Download Example Solution]

Exercise 6-9

In one sense, the Monkey and Banana programs are not good examples of Prolog plan generators, because in both versions the *ordering* of clauses is important, and has an effect on the types of plan generated. This is not in the spirit of (ideal) declarative programming, where a programmer should only be concerned with *describing* a problem accurately and not concerned with worrying about *how* it will run. What happens when the 'can_do' clauses in the original 'monkeyAndBanana.pl' program [view | download] or the 'place' clauses in the modified program from exercise 6-8 [view | download] are re-ordered in various ways - and why?

[Answer]

Exercise 6-10

Modify the answer to exercise 6-8 [view | download] so that the monkey can only climb onto the box if the box is in the middle of the room (imagine there is a step in the middle of the room), and so that there is an extra action of the monkey jumping off the box (wherever the box is). Imagine also that someone has initially lifted the monkey onto the box by the window. Test your answer to see if it can generate a plan for the monkey to get the banana from this new initial state, as well as from the previous initial state (where the monkey was by the door).

[View Example Solution | Download Example Solution]

Page maintained by Rob Miller. Last updated: 3/10/2019.
