Logical reasoning and programming, **lab session 10**
(December 4, 2023)

Task 1: Load the database of the British royal family from the last lab into Prolog.

```
female(elizabeth).
female(margaret).
female(diana).
female(camilla).
female(sophie).
female(kate).
female(meghan).
female(louise).

male(george).
male(philip).
male(charles).
male(edward).
male(william).
male(harry).
male(james).

parent(george,elizabeth).
parent(george, margaret).
parent(elizabeth,charles).
parent(philip,charles).
parent(elizabeth,edward).
parent(philip,edward).
parent(charles,william).
parent(diana,william).
parent(charles,harry).
parent(diana,harry).
parent(edward,louise).
parent(sophie,louise).
parent(edward,james).
parent(sophie,james).

wife(elizabeth,philip).
wife(diana,charles).
wife(camilla,charles).
wife(sophie,edward).
wife(kate, william).
wife(meghan, harry).
```

Task 2: Compare 4 implementations of the `ancestor(Anc,Desc)` predicate, which is supposed to connect the `Ancestor` with any of their descendants:

```
ancestor1(A,D) :- parent(A,D).
ancestor1(A,D) :- parent(A,B), ancestor1(B,D).

ancestor2(A,D) :- parent(A,D).
ancestor2(A,D) :- ancestor2(B,D), parent(A,B).

ancestor3(A,D) :- parent(A,B), ancestor3(B,D).
ancestor3(A,D) :- parent(A,D).

ancestor4(A,D) :- ancestor4(B,D), parent(A,B).
ancestor4(A,D) :- parent(A,D).
```

First without a computer, guess which implementation matches the behavior below. Please, verify your answers on a computer *after* you have made your guess.

a. Works as expected. Finds parents and children before grandparents and grandchildren.

b. Ends up in an infinite loop immediately.

c. Works as expected. Finds grandparents and grandchildren before parents and children.

d. Works as expected, but after the last response, Prolog ends up in an infinite loop.

Hint: Sketch an SLD tree for the query `ancestor`*`[1/2/3/4]`*`(X,charles)`.

Check your knowledge:

- What causes the infinite looping in the "bad" implementations?
- Would it help if *left-to-right* rule was changed to *right-to-left* rule?
- Or if the *top-to-bottom* rule was *bottom-to-top* rule?

Task 3: Match terms in various forms with their usual meaning.

Syntactic sugar	*Usual meaning*	*Hacker's syntax*
`[]`	list with 1 item	`'.'(X,Y)`
`[X,Y]`	empty list	`'.'(harry,[])`
`[harry]`	list with 2 items	`[]`
`[X|Y]`	list with at least 1 item	`'.'(X,'.'(Y,[]))`

Note: As of version 7, `SWI-Prolog` has changed the "list functor" from `'.'` to `'[|]'`. Keep it in mind if you wish to use the *hacker's syntax*. You may read up on that change here.

```
?- Z = '.'(harry, []).
ERROR: ...

?- Z = '[|]'(harry, []).
Z = [harry].

?- Z = '[|]'(X, Y).
Z = [X|Y].
```

Lesson learned: A list in Prolog is similar to structures in functional languages. If `L` is a list, then `[H|L]` is also a list, bigger by 1 element, where `H` is the element prepended to the beginning. You can add and/or remove elements from the beginning using the same syntax. `LL=[H|L]` both "adds" or "removes" the first element depending on whether `LL` or `L` are instantiated.

Task 4: Define basic predicates which work with lists:

- `list_of_size_one(X)` succeeds iff `X` has size exactly 1.
- `any_list(X)` succeeds iff `X` is any list.

  Hence `any_list(harry)` must fail, but `any_list([harry])` succeeds.
- `my_member(X, List)` succeeds iff `X` is inside the `List`.

  `my_member(b, [a,b,c])` must succeed and `my_member(d, [a,b,c])` must fail.

  If you did your implementation correctly, `my_member(X, [a,b,c])` should give you all 3 correct answers: `X=a; X=b; X=c.`

Task 5: Extend the (correct) implementation of `ancestor` so that in the 3rd argument, you get a list of people that are *between* the `Ancestor` and `Descendant`.

```
?- ancestor(diana, william, X).
X = [].

?- ancestor(A, william, X).
A = george,
X = [elizabeth, charles] ;
A = elizabeth,
X = [charles] ;
A = philip,
X = [charles] ;
A = charles,
X = [] ;
A = diana,
X = [].
```

See the beauty of Prolog! Ask for all great-grandparents:

`?- ancestor(GGParent,Person,[GParent,Parent]).`

Task 6: Define the `my_append(A, B, AB)` predicate.
It appends list `B` to the end of list `A` and gives the result in the third argument `AB`:

```
?- my_append(['micky mouse', 'donald duck', ted], [garfield, olaf], L).
L = ['micky mouse', 'donald duck', ted, garfield, olaf].
```

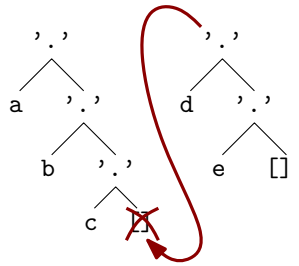
Does `my_append(L1, L2, [a, b, c])` provide you with (all) the correct answers?

Lost? See the hacker's syntax of lists `[a,b,c]` and `[d,e]`:

`'.'(a, '.'(b, '.'(c, [])))` and `'.'(d, '.'(e, []))`

Look carefully. Now, the secret to `append` is:

> If you remove `[]` in the *first* list and put the second list *instead*, you are done!



Still lost? What is the result of appending anything to an empty list?
Start from `my_append([],X,?)`...
The `my_append` should recurse on the first argument, strip one item after another until it reaches an empty list. Then it does the replacement.

Task 7: The main procedure behind Prolog is called *unification*. You have used it every time Prolog replaced a variable by a value (try `?-X=a`), but it's much more powerful. For example `[X,Y]= [a,b]` will *extract* values from inside the list.
The somewhat informal definition of unification is:

> Two terms unify if they can be made equal only by substituting variables.

For each of these terms, decide if they unify or not. If they unify, write down the substitution. Check your answers using Prolog. Has anything surprised you?

- `plus(X,Y) = plus(Z,4)`
- `'[|]'(first,'[|]'(second,[])) = '[|]'(A,'[|]'(B,[]))`
- `'[|]'(first,'[|]'(second,[])) = '[|]'(A,B)`
- `'[|]'(first,[]) = '[|]'(A,'[|]'(B,[]))`
- `'[|]'(X,'[|]'(Y,[])) = '[|]'(Y,'[|]'(element,[]))`
- `X = f(X)`
- `unify_with_occurs_check(X,f(X))`

Task 8: Define `my_reverse(List, Reversed)` that reverses elements in a list:

```
?- my_reverse([1,2,3,4,5], L).
L = [5,4,3,2,1].
```

Make sure Prolog does not end up in an infinite loop after returning the first answer!

Hint: Use `append(List, [X], ListX)` to add an element at the end of a list. Don't worry about time complexity. We will improve that soon enough!

Task 9: Define `minimum(List, Min)` which obtains a list a numbers and finds its minimum value.

Hint: Use an auxiliary predicate of arity three where one of the arguments stores the temporary result.

Task 10: Analogously to the temporary argument used in the last task, use an auxiliary argument to reduce the runtime of `my_reverse(List, Reversed)` from $\mathcal{O}(n^2)$ to $\mathcal{O}(n)$.

Hint: The auxiliary argument will be holding some part of the tail of the `Reversed` list.

Note: Such an argument is often called an *accumulator*.

Task 11: Extend the `my_member/2` predicate into the `my_select/3` predicate, which also gives the *rest* of the list:

```
?- my_select(Elem, [a,b,c], Rest).
Elem = a,
Rest = [b, c] ;
Elem = b,
Rest = [a, c] ;
Elem = c,
Rest = [a, b] ;
false.
```

Task 12 (optional): Rewrite the `ancestor/3` predicate so that the `Ancestor` and the `Descendant` are included in the list:

```
?- ancestor_all(X,Y,[P1, P2, P3, P4]).
X = P1, P1 = george,
Y = P4, P4 = william,
P2 = elizabeth,
P3 = charles ;
...
```

Task 13 (optional): Flatten a nested list.

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

It is enough to return additional (incorrect) answers. We'll learn how to remove them next week.