Practical Class 6
Database Modifications, Meta-Programming and Operators

Objectives:
- Making modifications to the database
- Using meta-logical predicates
- Declaring and using new operators

1. Family Relations Resurrections

Consider exercise 1 from the first exercise sheet, about family relationships.

a) Implement *add_person(+Gender, +Name)*, which adds a new person with the specified *Gender* (male/female) and *Name* to the knowledge base. If the person already exists, the predicate should fail.

b) Implement *add_parents(+Person, Parent1, Parent2)*, which adds the corresponding *parent/2* relations to the knowledge base. The predicate should only add non-existing relations, but always succeed (even if both relations already exist).

c) Implement *remove_person/0*, which prompts the user to select a person to remove from the knowledge base, removing them as well as any relationships in which the person is involved.

d) Implement *update_birthdates/0*, which prompts the user for the year of birth of every person in the knowledge base, replacing the *male/1* and *female/1* predicates by the updated version with two arguments (name and year of birth). Example:
```
| ?- update_birthdates.
What is the year of birth of jay?
|: 1946.
What is the year of birth of gloria?
|: 1972.
…
yes
| ?- male(X).
no
| ?- male(X,Y).
X = jay,
Y = 1946 ?
yes
```

e) Change the predicate above such that when the user inserts something other than an integer number, the system alerts the user and asks for the input again.

f) Change [some of] the predicates implemented in question 1.c from the first exercise sheet, such that different versions of the predicates are used depending on input instantiation for increased efficiency. Compare the original implementation with the new versions using the trace mode and different examples.

g) Implement *print_descendents(+Person)*, which, without using *findall*, *bagof*, or *setof*, prints to the terminal all children and grandchildren of *Person*. Example:

```
| ?- print_descendents(frank).
Children:
    phil
Grandchildren:
    haley
    alex
    luke
yes
```

Can you implement *print_descendents/1* without using other auxiliary predicates?

2. Higher-Order Predicates

a) Implement *map(+Pred, +List1, ?List2)*, with identical functionality to that of the *maplist/3* predicate, from the *lists* library. Example:

```
double(X, Y):- Y is X*2.

| ?- map(double, [2,4,8], L).
L = [4,8,16] ?
yes
```

b) Implement *fold(+Pred, +StartValue, +List, ?FinalValue)*, with identical functionality to that of the *scanlist/4* predicate from the *lists* library. Example:

```
sum(A, B, S):- S is A+B.

| ?- fold(sum, 0, [2, 4, 6], F).
F = 12 ?
yes
```

c) Implement *separate(+List, +Pred, -Yes, -No)*, which receives a list and a predicate, returning in *Yes* and *No* the elements *X* of *List* that make *Pred(X)* true or false, respectively.

```
even(X):- 0 =:= X mod 2.

| ?- separate([1,2,3,4,5], even, Y, N).
Y = [2,4],
N = [1,3,5] ?
yes
```

d) Implement *take_while(+Pred, +List, -Front, -Back)*, which identical functionality to that of the *group/4* predicate from the lists library. Example:

```
| ?- take_while(even, [2,4,6,7,8,9], F, B).
F = [2,4,6]
B = [7,8,9] ?
yes
```

e) Implement *ask_execute/0*, which reads a goal from the terminal and executes it. Example:

```
| ?- ask_execute.
Insert the goal to execute
|: map(double, [2,4,8], L), write(L).
[4,8,16]
yes
```

3. Functor, Arg, and Univ

a) Implement *my_functor/3*, with identical functionality to that of the *functor/3* predicate, using the =.. (univ) operator.

b) Implement *my_arg/3*, with identical functionality to that of the *arg/3* predicate, using the =.. (univ) operator.

c) Implement *univ/2*, with identical functionality to that of =.., based on the *arg/3* and *functor/3* predicates.

d) Define *univ/2* as an infix operator.

4. One Tree, Two Trees, Three Trees

Consider the definition of tree as seen in the lectures.

a) Implement *tree_size(+Tree, -Size)*, which determines the size of *Tree* (i.e., the number of nodes in the tree).

b) Implement *tree_map(+Pred, +Tree, ?NewTree)*, where *Tree* and *NewTree* are trees of the same shape, and *Pred(X, Y)* is true for every *X* in *Tree* and corresponding *Y* in *NewTree*.

c) Implement *tree_value_at_level(+Tree, ?Value, ?Level)*, which associates a value in *Tree* with the level of the tree in which that value exists; if only *Value* is specified but it does not exist in *Tree*, *-1* should be returned; if only *Level* is specified but no values exist at that level, the predicate should fail.

5. Operator Associativity

Consider the following operators:
```
:-op(500, xfx, na).
:-op(500, yfx, la).
:-op(500, xfy, ra).
```

Draw the parse tree for each of the following expressions.

a) a ra b na c

b) a la b na c

c) a na b la c

d) a na b ra c

e) a na b na c

f) a la b la c

g) a ra b ra c

h) a la b ra c

6. Operators

Consider the following operators:
```
:-op(550, xf, class).
:-op(560, xfx, of).
:-op(570, xfx, on).
:-op(560, xfx, at).
:-op(550, xfy, :).
```

Draw the parse tree for each of the following expressions.

a) `t class of pfl on tuesdays at 15.`

b) `tp class of pfl on mondays at 10:30.`

c) `pfl class at 17 on tuesdays of october.`

7. Creating Operators

Define operators that make the terms below syntactically valid.

```
a) flight tp1949 from porto to lisbon at 16:15

b) if X then Y else Z
```

Add the necessary code so that terms in the format of line b) are not only syntactically valid but also usable in code. Example:

```
| ?- if (3<4) then write(smaller) else write(greater).
smaller
yes
```

8. List operators

Define operators and the necessary relations so that it is possible to write and use terms such as:

a) `Element exists_in List`

b) `append A to B results_in C`

c) `remove Elem from List results_in Result`

9. Game of Nim

The game of Nim is played by two players. Initially, there are an arbitrary number of piles, each with an arbitrary number of matches. At each turn, a player takes one or more matches from one of the piles. The player to removes the last matches wins.

Implement *winning_move(+State, -Move)*, receiving the state (list of integers with the number of matches in each pile), returning a move (matches to remove and the pile from which to remove them), if any, guaranteed to win the game (whatever the opponent's moves in the following turns).

10. Turn Around

a) Implement *invert_predicate_clause_order/0*, which prompts the user for a predicate head, and inverts the order of the clauses in the code. Example

```
| ?- invert_predicate_cause_order.
Which predicate to you want to invert?
|: not(X).
yes
| ?- not(true).
true ? ;
no
```

b) Implement *insert_clause_with_inverted_goals/0*, which prompts the user for a clause, reads it from the terminal, and asserts it with the goals in reverse order. Example:

```
| ?- insert_clause_with_inverted_goals.
Insert a new predicate clause:
|: ola:- write(ola), write(mundo),write(adeus),write(mundo).
yes
| ?- ola.
mundoadeusmundoola
yes
```