Exercise 1 (Simple IO): (Not relevant for V3M! 5 points)

Please implement a predicate `starsquare/0` in Prolog which repeatedly asks the user to type in a number for the size of a square. If the user types in a positive number, the program displays a square consisting of stars (*) where one side of the square has as many stars as specified by the user. Below there are three squares for the inputs 1, 2 and 3. If the user types in a non-positive number, the program terminates. On all other inputs the program may behave arbitrarily.

\[
1: \quad 2: \quad 3:
\]

\[
* \quad ** \quad ***
\]

\[
** \quad * \quad *
\]

\[
***
\]

You may use the following code fragment as a starting point (but you do not have to use it):

```prolog
starsquare :- write('Size of square (0 for exit): '),
             read(S),
             nl,
             (drawsquare(S) -> nl,
              nl,
              starsquare
             ; true).
```

---

Hints:

- Please solve these exercises in **groups of two**!
- The solutions must be handed in **directly before (very latest: at the beginning of)** the exercise course on Wednesday, January 26th, 2011, in lecture hall **AH 3**. Alternatively you can drop your solutions into a box which is located right next to Prof. Giesl’s office (until the exercise course starts).
- Please write the names and **immatriculation numbers** of all (two) students on your solution. Also please staple the individual sheets!
- This whole exercise sheet is **not** relevant for students attending the **V3M (Master Informatik, SSE 6 credits)** version of the lecture. For these students this excluded exercise sheet also does not contribute to the overall maximum number of points that you can obtain.
- Since all exercises on this sheet are programming exercises, this time you are supposed to submit your solution **both written and by e-mail**. Please send your programs to lp10-hiwis@i2.informatik.rwth-aachen.de and write your names and matriculation numbers in the e-mail.
- The code fragments on this exercise sheet can also be downloaded from our webpage.
Exercise 2 (Advanced IO and program manipulation): (Not relevant for V3M! 10 points)

Please implement a predicate \texttt{neighborhood/0} in Prolog which shows the following prompt and asks the user for an input:

1: Add a new neighbor
2: Let a neighbor move away
3: Let a neighbor move within the neighborhood
4: Add an animosity between two neighbors
5: Show neighborhood
6: Show conflicts
7: Exit

Your choice:

Depending on the input of the user, the program shows the following behavior and then continues with the prompt from the beginning unless the input is the number 7. In this case, the program just terminates. For inputs from 1-6, the program does the following:

1 The program asks the user to type in the name of the new neighbor and the neighbor living right of him. If the latter is not existing in the neighborhood, the new neighbor is inserted into the neighborhood at the rightmost position. Otherwise the new neighbor is inserted into the neighborhood at the position being left of the specified right neighbor. We assume that the neighborhood is linear, i.e., every neighbor except the one living at the rightmost position has exactly one right neighbor. We also assume that every neighbor has a unique name. So if the user types in a name of a neighbor who already lives in the neighborhood, the program displays an error message. Finally, we assume that there is always enough space between two neighbors such that a new neighbor can move in between them.

2 The program asks the user for the name of the neighbor moving away from the neighborhood. If the specified neighbor does not exist in the neighborhood, the program displays an error message. Otherwise it deletes the specified neighbor from the neighborhood. If the neighbor being deleted was living at the rightmost position, his left neighbor now lives at that position. Otherwise the right neighbor of the deleted neighbor becomes the right neighbor of the deleted neighbor’s left neighbor. For example, if \texttt{anna}, \texttt{boris} and \texttt{charly} live together in the neighborhood where \texttt{boris} lives right of \texttt{anna} and \texttt{charly} lives right of \texttt{boris} and \texttt{boris} moves away, then \texttt{charly} becomes the right neighbor of \texttt{anna}. Additionally, all knowledge about animosities where the deleted neighbor is part of the animosity is deleted.

3 The program asks the user for the name of the moving neighbor and his new right neighbor. If the specified moving neighbor does not exist in the neighborhood, the program displays an error message. Otherwise the specified moving neighbor moves to the position being left of the specified right neighbor. If the latter does not exist in the neighborhood, the specified moving neighbor moves to the rightmost position. It is allowed that a neighbor moves to the position where he already is.

4 The program asks the user to type in the two names of the neighbors having the animosity (i.e., they dislike each other). If any of them does not exist in the neighborhood or if they already have an animosity, the program displays an error message. Otherwise it adds the knowledge about the animosity between the two specified neighbors. We assume that having an animosity is always symmetric. For example, if we already know that \texttt{anna} and \texttt{boris} have an animosity and the user wants to add an animosity between \texttt{boris} and \texttt{anna}, the program displays an error message.

5 The program displays all neighbors living in the neighborhood separated by arrows pointing to their respective right neighbor. For example, if the current neighborhood consists of the four neighbors \texttt{anna}, \texttt{boris}, \texttt{charly} and \texttt{diana} where \texttt{boris} lives right of \texttt{anna}, \texttt{charly} lives right of \texttt{boris}, and \texttt{diana} lives at the rightmost position, the output of the program is:

\texttt{anna --> boris --> charly --> diana}
6 The program displays all conflicts in the neighborhood. A conflict consists of two neighbors who dislike each other and live next to each other. The two conflicting neighbors are displayed separated by a star (•), while several conflicting pairs of neighbors are displayed one below another (the order of the conflicting pairs of neighbors is up to your implementation). Symmetric conflicts are displayed only once (the order of the neighbors in the conflicting pairs is also up to your implementation). In the example from above, we add knowledge about animosities between anna and boris, boris and charly, and anna and diana. Then the output of the program is (modulo different orderings):

boris•charly
anna•boris

For any other input than the numbers 1-7 the program displays an error message. To solve this exercise, you can for example use dynamic predicates which you assert and retract. Alternatively, you can use input and output to store information about neighbors and animosities in files. If you want to use assert and retract, you may use the following code fragment as a starting point (but you do not have to use it):

```prolog
:- dynamic(neighbors/1).
:- dynamic(animosity/2).
neighbors([]).
neighborhood :- showMenu, read(X), nl, (X = 7 -> true ; choice(X), nl, nl, neighborhood).
showMenu :- write('1: Add a new neighbor'), nl, write('2: Let a neighbor move away'), nl, write('3: Let a neighbor move within the neighborhood'), nl, write('4: Add an animosity between two neighbors'), nl, write('5: Show neighborhood'), nl, write('6: Show conflicts'), nl, write('7: Exit'), nl, nl, write('Your choice: ').
```

Exercise 3 (Term manipulation): (Not relevant for V3M! 5 points)

You already know the notion of unification, i.e., two terms \( t_1 \) and \( t_2 \) unify iff there is a substitution \( \sigma \) such that \( \sigma(t_1) = \sigma(t_2) \). A similar notion is matching. A term \( t_1 \) matches a term \( t_2 \) iff there is a substitution \( \mu \) such that \( \mu(t_1) = t_2 \). The substitution \( \mu \) is then called the matcher of \( t_1 \) and \( t_2 \).
Please implement a predicate `match/3` in Prolog where `match(t1,t2,t3)` works as follows. If `t1` does not match `t2`, the predicate fails. Otherwise `t3` is unified with a list representing the smallest matcher of `t1` and `t2`, i.e., the matching substitution which replaces as few variables as possible. However, the evaluation of `match/3` just computes this matcher without actually replacing variables in `t1` (unless they also occur in `t3`). For example, the evaluation of `?- match(f(X,Y),f(0,1),S).` succeeds with the answer substitution `S = [X / 0, Y / 1]` (and not with the answer substitution `X = 0, Y = 1, S = [0 / 0, 1 / 1]`), while the evaluation of `?- match(f(X,Y),g(0,1),S).` fails.

**Hint:** Remember that `/` is an infix operator by default. Built-in predicates for term manipulation like `var/1, ==/2` or `=/2` can be useful for your algorithm.

**Exercise 4 (Difference Lists):**  
(Not relevant for V3M! 2 points)

Please implement a predicate `toDL/2` in Prolog which computes the most general difference list representation for a given list, i.e., a query `?- toDL([q1,...,qn],XS).` yields the answer substitution `XS = [q1,...,qn|ZS]-ZS`, where `ZS` is a fresh variable.