CSCI 235, Programming Languages, Assignment 4

Rules for Assignments:

- You are expected to read the complete assignment and to complete every part of it. 'I did not see this part' will never be a valid excuse for not completing a part of the assignment.
- Submitted code will be checked for correctness, readability, style, and layout. We apply high quality standards during grading.
- If you need help, please post your question on Piazza. Questions that do not contain code can be asked publicly. Questions that contain code must be posted as private question to all instructors. If you are in doubt, post as private question, because instructors can always make the question public later. Don't mail questions directly to an instructor, because individual instructors are not always available. Piazza is watched almost full time, so you will get a quicker answer.
- You must write all submitted code by yourself! We check code for similarity using automatic tools and also by visual inspection. See the syllabus for additional rules. You may be invited for live grading, or for discussion about the quality of your code.

Introduction

Goal of this assignment is to make you familiar with Prolog as a language in which real programs can be written, and convince you that search is easy in Prolog. Before starting, read the remarks that follow. From my experience with teaching Prolog, these are the things that students have problems with:

1. Since Prolog has no **while** or **for** loops, the only way by which things can be repeated, is recursion.
2. Prolog has no way of evaluating functions. If you write `f(a,2)`, this denotes a tree with `f` on top, and two subtrees, `a` and `2`.
3. Prolog is based on *logic*, instead of a machine model. Variables can be instantiated, but not reassigned, because that would be logically incorrect.

A query of form `X is X + 1` will always fail, because there is no number for which $X = X + 1$.

4. Don't use double equality `==`. Students seem to like it, I do not know why. Read the slides and watch the lecture videos instead of believing a random stranger on the internet. The `==` predicates compares two terms without trying to make them equal (and thus it breaks the logical core of Prolog). You can see the difference by typing `X == 1` and `X = 1`. In case the reason don't convince you, note that using `==` will cost you points.

5. Don't use the cut predicate (!). It is sometimes useful, but in my experience use by students is a symptome not grasping the logical foundation of Prolog. Using this predicate will result in loss of points.

Tasks

1. Write a predicate `cartesian( X, Y, Z )` that is true if `X,Y,Z` are lists, and `Z` is the Cartesian product of `X` and `Y`, i.e. a list of all possible pairs of members of `X` and `Y`. The predicate must be able to compute `Z`, when `X` and `Y` are given:

   ```
   cartesian( [], [1], Z )      %  gives Z = []
   cartesian( [a], [], Z )      %  gives Z = []
   cartesian( [a], [b], X )     %  gives Z = [pair(a, b)]
   cartesian( [a,b], [1,2], Z )
      %  gives Z = [pair(a, 1), pair(a, 2), pair(b, 1), pair(b, 2)].
   ```

   For the use of `pair`, see remark 2 above.

   In order to solve this task, you will need a helper predicate `makepairs(X,Y,Z)` that is true if `Y` is a list, and `Z` is obtained from `Y` by replacing every element `E` of `Y` by `pair(X,E)`. Use recursion on `Y`.

   Once you have `makepairs`, you can write `cartesian(X,Y,Z)` by recursion on `X`. In the implementation of `cartesian`, you may use `append`. In the implementation of `makepairs`, you don't need it.

2. Since Prolog is untyped, it is possible to mix various kinds of elements in a list, also elements with different levels of nesting, like for example `[ 1, a, [ b ], [ c, [ 3 ] ] ]`. Write a predicate `deepsum( X, Y )` that succeeds if `X` is a possibly nested list, and `Y` is the sum of the numbers occurring in `X`. `deepsum` must ignore everything that is not a number.

   ```
   deepsum( [ [ 2, a ], [3], "hello" ], Y ).
      %  results in Y = 5.
   deepsum( [ [ b, c ], "X" ], Y ).
      %  results in Y = 0.
   deepsum( [ [ 2, 1 ], [[[5]]] ], Y ).
      %  results in Y = 8.
   ```

You can use the following predicates: `num(X)` succeeds when `X` is a number. `atom(X)` succeeds when `X` is an atom, and `string(X)` succeeds when `X` is a string.

3. Modify `deepsum` in such a way that it separates the sums of the negative and the positive numbers as follows:

```
deepsum( [ [ 2, a ], [-3], "hello" ], Pos, Neg ).
   %  results in Pos = 2, Neg = -3.
deepsum( [ [ b, c ], "X" ], Pos, Neg ).
   %  results in Pos = 0, Neg = 0.
deepsum( [ [ 2, -1 ], [[[-5],3]] ], Pos, Neg ).
   %  results in Pos = 5, Neg = -6.
```

   You can reuse the name `deepsum` because Prolog allows predicate overloading by arity.

4. Since Prolog is good at search, we give a task involving search. A *magic square* of size n is an $n \times n$ matrix which contains each number between 1 and n^2 exactly once, and in addition, the sums of all rows, all columns, and the two diagonals are equal. Below are a magic square of size 3 and a magic square of size 4 :

8	1	6
3	5	7
4	9	2

16	3	2	13
5	10	11	8
9	6	7	12
4	15	14	1

   We are going to generate magic squares of size 4 by exhaustive search.

   (a) Write a predicate `numbers(L)` that is true if `L` is the list of numbers that can be used in the magic square. (It consists of a single fact.)

   (b) Write a predicate `select(X,L1,L2)` that succeeds if `X` occurs in `L1`, and `L2` is obtained from `L1` by removing `X`. The first and the last argument can be uninstantiated, but the middle argument is always instantiated. For example,

```
select( 2, [ 2, 3 ], X )    % results in X = [3].
select( X, [ 2, 3 ], R )
   % X = 2,  R = [3].
   % X = 3,  R = [2].
select( 3, [ 5, 6 ], R ).   % fails.
```

   (c) Write a predicate `sumfits( N1, N2, N3, N4 )` that succeeds when `N1 + N2 + N3 + N4` equals 34. It must be possible to call `sumfits` when `N4` is uninstantiated.

   (d) Create the predicate

```
start( S, Free16 ) :-
S = [ [ A11, A12, A13, A14 ],
      [ A21, A22, A23, A24 ],
      [ A31, A32, A33, A34 ],
      [ A41, A42, A43, A44 ] ],
numbers( Free16 ).
```

We will create a sequence of predicates that succeed when the variables have distinct values between 1 and 16, and all the required sums are equal to 34. The interpreter will use these predicates to find values for the variables by try and error. If we implement this not carefully, the interpreter will create $16! \approx 2.09 \times 10^{13}$ instantiations, which is not feasible. For example, the following approach would not work:

```
numbers(Free16),
select(A11,Free16,Free15), select(A12,Free15,Free14),
select(A13,Free14,Free13), ..., select(A44,Free1,Free0),
Free0 = [], ...  % do the checks here, way too late.
```

Enumerating all permutations can be avoided by checking the sums as early as possible.

- As soon as we have guessed values `A11,A12,A13,A14`, we can check the sum.
- Moreover, as soon as we have guessed values for `A11,A12,A13`, we can compute the value for `A14` with the `sumfits` predicate, and check that this value is stil available.
- Once we have generated and checked `A11,A12,A13,A14`, we can generate and check the next row. However, instead of generating `A21,A22,A23,A24`, it is better to generate `A21,A31,A41`, because then we have column `A11,A21,A31,A41` complete, which we can check. In order to do this, we need to guess two values and verify one value.
- At this point, we should generate `A32` and check `A23`. This allows us to check the diagonal `A41,A32,A23,A14` while guessing only a single number.
- One also needs to consider symmetries: If one rotates or mirrors a magic square, the result is still a magic square. There is no point in generating essentially the same square eight times, so we will require that `A11` is smaller than the three other corners (this solves the rotation problem), and we will require that the left bottom corner is smaller than the top right corner `A41 < A14`. (This solves the mirroring problem.)

(e) Write a predicate `printsquare( S )` that prints magic square `S` in a readable fashion. Decompose `S` into `[ R1, R2, R3, R4 ]`, and use `write` and `nl` to print the rows. If everything goes well, then the query `start( S, Free16 ), printsquare(S)` prints

```
[_676,_682,_688,_694]
[_706,_712,_718,_724]
[_736,_742,_748,_754]
[_766,_772,_778,_784]
S = [[_, _, _, _], [_, _, _, _], [_, _, _, _], [_, _, _, _]],
Free16 = [1, 2, 3, 4, 5, 6, 7, 8, 9|...].
```

(f) Create a predicate

```
step1( S, Free7 ) :-
   S = [ [ A11, A12, A13, A14 ],
         [ A21, A22, A23, A24 ],
         [ A31, A32, A33, A34 ],
         [ A41, A42, A43, A44 ] ],
   start( S, Free16 ), ...
```

that creates values for `A11,A12,A13,A14,A21,A31,A41,A32,A23`, and checks that they are correct. It must work as follows:

- Generate values for `A11,A12,A13` using `select()`. Generate the value for `A14` using `sumfits( )`. Check that `A14` is available using `select( )`, and check that `A11 < A14`.
- Generate values for `A21,A31` using `select( )`. Generate the value for `A41` using `sumfits( )`. Check that `A41` is available using `select( )`, and check `A11 < A41, A41 < A14`.
- Generate `A23` using `select( )`, generate `A32` using `sumfits( )` and check it with `select( )`.

This diagram shows the part of the square that `step1` instantiates:

X	X	X	X
X		X	
X	X		
X			

(g) Create a predicate

```
step2( S, Free4 ) :-
S = [ [ A11, A12, A13, A14 ],
      [ A21, A22, A23, A24 ],
      [ A31, A32, A33, A34 ],
      [ A41, A42, A43, A44 ] ],
step1( S, Free7 ), ...
```

that finds values for `A22,A24,A42`. Use the predicates `select` and `sumfits` in optimal fashion. The effect of `step2` is as follows:

X	X	X	X
X		X	
X	X		
X			

$\Rightarrow$

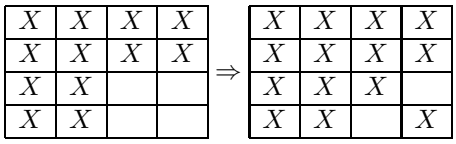
X	X	X	X
X	X	X	X
X	X		
X	X		

(h) Write a predicate

```
step3( S, Free2 ) :-
   S = [ [ A11, A12, A13, A14 ],
         [ A21, A22, A23, A24 ],
         [ A31, A32, A33, A34 ],
         [ A41, A42, A43, A44 ] ],
   step2( S, Free4 ), ...
```

that instantiates `A33,A44`. In order to do this, guess a value for `A33` using `select( )`, compute the value for `A44` using `sumfits( )` and verify that this value is still available using `select( )`. Also check that `A11 < A44`.

Below is the effect of `step3`:



(i) Write the predicate

```
step4( S, Free0 ) :-
   S = [ [ A11, A12, A13, A14 ],
         [ A21, A22, A23, A24 ],
         [ A31, A32, A33, A34 ],
         [ A41, A42, A43, A44 ] ],
   step3( S, Free4 ), ...
```

that instantiates the final positions `A43,A34`. Both positions should be generated with `sumfits( )` and checked with `select( )`. In order to generate all magic squares, you can use:

```
printall :-
   step4( S, Free0 ),
   Free0 = [],
   printsquare(S), fail.
```

If everything goes well, 880 will be printed.

Submit a single text file called **assignment4.pl**. Don't use an archiver!