CS 403, Assignment 3

Due on 7 November at 11:59 pm

We will now try our hands at the somehow ambitious goal of solving a nontrivial kind of a puzzle in PROLOG.

For this purpose, an image consists of $n \times m$ pixels, where each pixel can be either black or white. We receive a description of such an image that identifies the groups of black pixels in each row and each column of the image as follows: A group is identified by the number of consecutive black pixels therein (that is, an integer). Each row or column is then labeled with the list of black groups (that is, integers) therein. The black groups are given left to right for rows, and top to bottom for columns.

Here is an example of such an image together with its numerical representation described above. The number in the column lists are given one on top of the other.

□■■■□□□□3
■■□■□□□□2, 1
□■■■□□■■3, 2
□□■■□□■■2, 2
□□■■■■■■6,
■□■■■■■□1, 5
■■■■■■□□6,
□□□□■□□□1
□□□■■□□□1
1 3 1 7 5 3 4 3
2 1 5 1

Your overall mission is to reconstruct the image from its numerical description. Your input data is given as two lists, one for the rows and the other for the columns in the image. Each list contains lists of numbers (the black groups). For example, the image above is described by the following two lists:

```
[[3],[2,1],[3,2],[2,2],[6],[1,5],[6],[1],[2]]
[[1,2],[3,1],[1,5],[7,1],[5],[3],[4],[3]]
```

The output should then be a list of rows, where each row consists of a list of symbols `' '` (for a white pixel) and `x` (for a black pixel). For example the image above will be represented by the following list:

```
[[' ', x , x , x ,' ',' ',' ',' ',' '],
 [ x , x ,' ', x ,' ',' ',' ',' ',' '],
 [' ', x , x , x ,' ',' ', x , x ],
 [' ',' ', x , x ,' ',' ', x , x ],
 [' ',' ', x , x , x , x , x , x ],
 [ x ,' ', x , x , x , x , x ,' '],
 [ x , x , x , x , x , x ,' ',' '],
 [' ',' ',' ',' ', x ,' ',' ',' '],
 [' ',' ',' ', x , x ,' ',' ',' ']]
```

The following questions will guide you toward a solution:

1. Write a predicate `generateRow/2` such that `generateRow(H,L)` will bind `L` to a list of length `H` consisting of an arbitrary number of `x`'s and blanks. Here is a sample query:

```
?- generateRow(3,L).
L = [' ', ' ', ' '] ;
L = [' ', ' ', x] ;
L = [' ', x, ' '] ;
L = [' ', x, x] ;
L = [x, ' ', ' '] ;
L = [x, ' ', x] ;
L = [x, x, ' '] ;
L = [x, x, x] ;
false.
```

   Now go one step further with the predicate `generateRowCheck/3` with one extra argument that specifies the runs of `x`'s in the list. Accordingly, `generateRowcheck(Runs,H,Row)` generates a list just like `generateRow(H,Row)`, in which additionally the length of the runs of `x`'s are given by `Runs` (which is a list of integers). Here are a couple of sample queries:

```
?- generateRowCheck([1,1],3,L).
L = [x, ' ', x] ;
false.

?- generateRowCheck([2],3,L).
L = [' ', x, x] ;
L = [x, x, ' '] ;
false.
```

   Note that in the first example there is a single possible row that has two runs of one `x` each, whereas a run of two `x`'s can happen in two ways.

2. Extend now the predicate developed above to more than one row: Define the predicate `generateImage/4` such that `generateImage(V,H,Runs,Image)` binds `Image` to a list of `V` rows, each row having the length `H`. Furthermore the runs of `x`'s in the row must observe the restriction on their lengths given by the corresponding value in `Runs` (which is therefore a list of length `V` of lists of integers). A query that uses this predicate will go like this:

```
?- generateImage(4,3,[[2],[1,1],[1],[3]],I).
I = [[' ', x, x], [x, ' ', x], [' ', ' ', x], [x, x, x]] ;
I = [[' ', x, x], [x, ' ', x], [' ', x, ' '], [x, x, x]] ;
I = [[' ', x, x], [x, ' ', x], [x, ' ', ' '], [x, x, x]] ;
I = [[x, x, ' '], [x, ' ', x], [' ', ' ', x], [x, x, x]] ;
I = [[x, x, ' '], [x, ' ', x], [' ', x, ' '], [x, x, x]] ;
I = [[x, x, ' '], [x, ' ', x], [x, ' ', ' '], [x, x, x]] ;
false.
```

3. Now that we have an image let's transpose it: Define the predicate `transposeImage/2` such that `transpose(Image,Result)` receives as first argument an image (list of rows of `x`'s and blanks) and then binds `Result` to a list of all the `columns` in `Image`. Something like this:

```
?- transposeImage([[x, x, ' '], [x, ' ', x], [x, ' ', ' ']], I).
I = [[x, x, x], [x, ' ', ' '], [' ', x, ' ']] ;
false.
```

4. Now put everything together to solve the puzzle. Provide a predicate that receives the two list of constraints (horizontal and vertical) and binds a third argument to an image compatible with those constraints. You would use this predicate like this:

```
?- solve_puzzle([[3],[2,1],[3,2],[2,2],[6],[1,5],[6],[1],[2]],
                [[1,2],[3,1],[1,5],[7,1],[5],[3],[4],[3]], N).
```

   It has been claimed that this kind of puzzles has at most one solution, though to the best of my knowledge this has never been proven.

5. Note that it takes quite a bit of time to solve such a puzzle. My implementation required 8 million inferences to solve the puzzle provided in this handout and a further 1.7 million inferences to fail afterward (thus ensuring that no other solution is possible). On my relatively beefy I7 CPU that took about 10 minutes for the solution and a further 2-3 minutes to ensure uniqueness.

   For a bonus, can you think of a more efficient way to solve this kind of puzzles? A sketch solution would be acceptable as long as it is sketched in something that resembles PROLOG; a complete, working solution would be even better.

   Note in passing that the puzzles solved by enthusiasts are much larger than the one used above as an example. As a case in point, here is a smallish such a puzzle, with `H` the black runs of the row and and `V` the black runs of the columns:

```
H = [[2], [4], [4], [3,4], [9], [8], [7], [8], [9], [3,4], [3],
     [3], [3], [3], [3], [3], [3], [3], [3], [3], [3], [3], [3],
     [5], [3], [7], [7], [5], [10], [13], [14], [4,4,3], [3,2],
     [2,1,2], [2,3,3,2], [2,4,3,2], [3,3,2,3], [6,6], [4,4], [2,2],
     [2,2,2], [8], [6], [4], [2]]

V = [[6], [8], [4,3], [4,2], [3,2,5], [2,3,10], [2,1,7,4,3],
     [32,2,5], [32,5], [31,4,3], [5,1,6,5,2], [5,2,3,2,5],
     [7,3,4], [7,3,3], [3,3,9], [2,2,6]]
```

   Getting a solution to this puzzle using the approach outlined above is going to take hours, which illustrates quite vehemently the poor performance of this solver. There *must* be a better solution to this.

Implementation note and helpful predicates

PROLOG being an imperfect implementation of the predicate calculus is a rather messy programming language, but it does not have to be that way. I require that you keep as close to the predicate calculus as possible; remember that our goal is primarily to talk about the principles rather than the practice of logic programming. In particular, I have not talked and will not talk about inference control such as the cut predicate. Therefore using inference control in your implementation will be severely penalized (you are not supposed to know about that anyway).

The length of a list can be obtained using the predefined predicate `length/2` which receives a list as its first argument and binds its second argument to the length of that list.

You will notice that the lists being produced by your code are printed by SWI Prolog in a truncated manner. It would also be nice to print the solutions of the puzzle in a more visual manner. Therefore you may want to use the following predicate (which you will have to copy in your program since it is not predefined anywhere):

```
writelist([]).
writelist([H|T]) :- write(H), nl, writelist(T).
```

One way to use this predicate is in conjunction with the main query that solves the puzzle, something like this:

```
?- solve_puzzle([[3],[2,1],[3,2],[2,2],[6],[1,5],[6],[1],[2]],
                [[1,2],[3,1],[1,5],[7,1],[5],[3],[4],[3]], Result),
   writelist(Result).
```

Submission guidelines

Submit a single plain text file suitable for consulting from within the PROLOG interpreter and containing the non-programming deliverables (such as your sketch of an alternate solution) as comments. Also provide suitable comments that demonstrate and test your solution, such as copy-and-paste content from the terminal where you ran and tested your program. Any part of your work that has not been thus demonstrated will not be marked.

Note that in PROLOG the character % is used as comment prefix, but C-style block comments (`/* ... */`) are also available.

Submit your script by email. Do not forget to include at the top of your script a comment with the names and emails of all the collaborators.

Recall that assignments can be solved in groups (of maximum three students), and a single solution per group should be submitted. Also recall that a penalty of 10% per day will be applied to late submissions.