Knowledge representation (INFO0049-1)
Exercise session 3
3 March 2015

Try to draw search trees wherever possible to see how prolog executes a query

---

Exercise on Lists

Basic list-predicates

1. Define predicates for

- Retrieving the first element of a list: first(+List, -Element)
- Retrieving the last element of a list: last(+List, -Element)
- Returning the length of a list: list_length(+List, -Length)
- Removing in a list ALL elements equal to T: remove_all(+T, +List, -NewList)
- Removing in a list one element equal to T: remove(+T, +List, -NewList)
- Removing in a list of numbers all elements smaller than or equal to N: smaller(+N, +NumberList, -NewNumberList)
- Switching the first two elements of a list: switch_first_two(+List, -NewList). For example a call to switch_first_two([a,b,c,d,e,f], L) should return L = [b,a,c,d,e,f]
- Switching every pair of elements in a list: switch_every_two(+List, -NewList). For example a call to switch_every_two([a,b,c,d,e,f,g], L) should return L = [b,a,d,c,f,e,g]

2. Write a predicate calculate_sum(+List, -Sum) that, given a list of numbers, calculates the sum of all these numbers. Then write a predicate calculate(+List, -Avg) that calculates the average of the numbers in the list. Can you do this by traversing the list only once?

3. LOA is a game comparable to checkers. The board consists of squares. Each square can be filled with a white or black disk, or be empty. Suppose that we represent a row on the board as a list where each element is either the atom w (white disk), the atom b (black disk) or the atom n (no disk). Define

the predicate count_disk(+Row, -Count) that given a row returns the number of disks on that row. What is the answer to the following queries?

?- count_disk([w,b,n,n],Count).
?- count_disk([w,b,n,n],2).
?- count_disk([w,b,n,n],4).
?- count_disk(List,Count).

4. We store a LOA game as a list of all moves (in chronological order). A move (starting from one position on the board, going to another position) is represented with a move functor with 4 arguments: the first and second arguments are the column (a letter) and row (a number) of the from-position; the third and fourth arguments are the column and row of the to-position. E.g. we can have the following game: [move(b,1,b,3), move(c,6,c,8), move(b,3,b,5)]. In this game the first player moved from b1 to b3, then the second player moved from c6 to c8 and then the first player moved from b3 to b5.

1. Write a predicate nb_rounds(+List, +NoOfRounds) that, given a movelist, calculates the number of 'rounds' played in the game. With 'round' we mean one move of each of the two players (unless at the end of the game).

   ?- nb_rounds([move(b,1,b,3), move(c,6,c,8), move(b,3,b,5)], Count).

   Count = 2
   true.

2. Write a predicate splits(+List, -List1, -List2) that splits a given movelist (List) into two lists: one with the black moves (List1), the other with the white moves (List2). Keep in mind that a movelist can contain an odd number of moves (like in the movelist given above).

   ?- split([move(b,1,b,3), move(c,10,c,8), move(b,3,b,5)], BlackMoves, WhiteMoves).

   BlackMoves = [move(b,1,b,3), move(b,3,b,5)]
   WhiteMoves = [move(c,10,c,8)]
   true.

3. Write a predicate merge(+List1, +List2, -List) that given a list with the moves from black (List1) and a list with the moves from white merges (List2) both into a chronological movelist (List).

5. Write a predicate cattree(+N,-Tr), where N is a natural number and Tr is a binary tree with N internal notes, and leaves labeled with

symbol x. General all possible binary trees with N internal nodes (N+1 leaves).

?- cattree(4,Tr).

Tr = [x, [x, [x, [x, x]]]] ;
Tr = [x, [x, [[x, x], x]]] ;
Tr = [x, [[x, x], [x, x]]] ;
Tr = [x, [[x, [x, x]], x]] ;
Tr = [x, [[[x, x], x], x]] ;
Tr = [[x, x], [x, [x, x]]] ;
Tr = [[x, x], [[x, x], x]] ;
Tr = [[x, [x, x]], [x, x]] ;
Tr = [[[x, x], x], [x, x]] ;
Tr = [[x, [x, [x, x]]], x] ;
Tr = [[x, [[x, x], x]], x] ;
Tr = [[[x, x], [x, x]], x] ;
Tr = [[[x, [x, x]], x], x] ;
Tr = [[[[x, x], x], x], x] ;
false.

6. Given a list of lists write a predicate flatten_list_of_lists(+List, -Flatlist) that `flattens' the list.
?- flatten_list_of_lists([[a, b, c], [d, e, f]], FlatList).

FlatList = [a,b,c,d,e,f]
true.

7. Suppose we represent a matrix as a list of lists, e.g.:

A = [[a11,a12,a13], [a21,a22,a23]]

Write a predicate find_transpose(+A, -AT) that computes the transposed matrix:

AT = [[a11,a21], [a12,a22], [a13,a23]]

Your code should work for any matrix, not only for a 2-by-3 matrix.
