CS-UY 3314 — Homework 10

Jeff Epstein

Introduction

Complete the following exercises in Prolog.

Before starting, please make sure that you're using a suitable programming environment, as discussed in previous homeworks.

For all questions in this assignment, follow the instructions carefully. You may not use any modules not loaded by default, and you may not use any extensions to the language (such as DCG or CLP(X)), except as specified. You may not use the `->` and `;` operators, unless otherwise specified.

Submission

Unless otherwise specified, put your solutions into a file named `hw`n`.pl`, where n is the number of this assignment.

Test your code thoroughly before you submit. The test cases provided should guide you in verifying your solution.

Make sure that you name your predicates and/or facts exactly with the names specified in the assignment. For questions whose solution is not code, provide the answer within a comment. In addition, mark each solution with a single-line comment indicating the question number. Single-line comments in Prolog begin with percent-sign (`%`).

Make sure that each file is syntactically correct and can be compiled. If one of your solutions is partial or incomplete, please comment it out (using `/*` and `*/`) so that the remainder of the file can be compiled. If we can't compile your file, we can't grade it, and your grade may be penalized.

Submit all your solution files on Gradescope.

1 Trees

1. We will represent a binary tree using terms of the form `t(V,Left,Right)`, where `V` stores the value of the node, and where `Left` and `Right` store terms for the left and right subtrees. An empty tree will be represented with the term `nil`. For example, the following predicate will unify its parameter with a binary tree of depth three:

```
someTree(X) :-
  X = t("t1",
    t("t2a",
      t("t3a", nil, nil),
      t("t3b", nil, nil)),
    t("t2b",
      t("t3c", nil, nil),
      t("t3d", nil, nil))).
```

We can write a predicate `flipNode/2` such that `flipNode(A,B)` will flip the left and right children of a given node `A`, so that the output `B` is a node whose left child is the right child of the input node, and whose right child is the left child of the input node. Attempting to flip an empty tree just produces another empty tree, since there is nothing to flip.

```
flipNode(nil, nil).
flipNode(t(V, L, R), t(V, R, L)).
```

Using the `flipNode/2` predicate, we can issue the following query:

```
?- someTree(Unflipped), flipNode(Unflipped, Flipped).
Unflipped = t("t1",
  t("t2a", t("t3a", nil, nil), t("t3b", nil, nil)),
  t("t2b", t("t3c", nil, nil), t("t3d", nil, nil))),
Flipped = t("t1",
  t("t2b", t("t3c", nil, nil), t("t3d", nil, nil)),
  t("t2a", t("t3a", nil, nil), t("t3b", nil, nil))).
```

Write a recursive predicate `flipTree/2`, which will produce a mirror image of the whole tree. For example:

```
?- someTree(Unflipped), flipTree(Unflipped, Flipped).
Unflipped = t("t1",
  t("t2a", t("t3a", nil, nil), t("t3b", nil, nil)),
  t("t2b", t("t3c", nil, nil), t("t3d", nil, nil))),
Flipped = t("t1",
  t("t2b", t("t3d", nil, nil), t("t3c", nil, nil)),
  t("t2a", t("t3b", nil, nil), t("t3a", nil, nil))).
```

When we write Prolog code that can produce multiple values as results, there are typically two approaches:

- the *spatial representation* — multiple results are returned in a single list. This is similar to how you might write a program in, for example, Haskell.

  It's called spatial because each result is returned at the same time, and therefore the results differ in the *space* (i.e. memory location) they exist.

  For example, consider this predicate:

  ```
  countSpatial(0,[]).
  countSpatial(N,[N|Tail]) :- N>0, N2 is N-1,
      countSpatial(N2, Tail).
  ```

  which produces this output:

  ```
  ?- countSpatial(4, X).
  X = [4, 3, 2, 1] ;
  false.
  ```

Read and understand the above predicate.

- the *temporal representation* — multiple results are returned as a sequence of unifications. This approach is unique to languages with backtracking.

  It's called temporal because each result is returned one after another, and therefore the results differ in the *time* they exist.

  For example, consider this predicate:

```
countTemporal(N,N) :- N>0.
countTemporal(N,X) :- N>0, N2 is N-1, countTemporal(N2, X).
```

  which produces this output:

```
?- countTemporal(4,X).
X = 4 ;
X = 3 ;
X = 2 ;
X = 1 ;
false.
```

  Read and understand the above predicate.

  Note that although the values produced in the temporal version are the same as in the the spatial version, they differ in how they are presented: as a single list of numbers (spatial), or as a sequence of unifications with numbers (temporal).

In the exercises 2 and 3, you may not use the predicates `findall`, `bagof`, `member`, and similar.

2. Let's start with a spatial representation. Write a recursive predicate `inorderTraversalSpatial/2` that will produce a list of values from an in-order traversal of a tree. For example:

```
?- someTree(ATree), inorderTraversalSpatial(ATree, AList).
ATree = t("t1",
  t("t2a", t("t3a", nil, nil), t("t3b", nil, nil)),
  t("t2b", t("t3c", nil, nil), t("t3d", nil, nil))),
AList = ["t3a", "t2a", "t3b", "t1", "t3c", "t2b", "t3d"].
```

   Note that `AList` unifies with a single value, i.e. the list containing each of the nodes of the tree. This is therefore a spatial representation of the answer.

   Make sure that your solution correctly handles empty trees and unbalanced trees.

   Hint: use the built-in `append/3` predicate, which works like this:

```
?- append([a,b,c], [d,e,f], X).
X = [a, b, c, d, e, f].
```

3. In contrast to the previous question, let's now tackle the temporal representation. Write a recursive predicate `inorderTraversalTemporal/2` that will produce a sequence of results from an in-order traversal of a tree. For example:

```
?- someTree(ATree), inorderTraversalTemporal(ATree, Result).
ATree = omitted,
Result = "t3a" ;
ATree = omitted,
Result = "t2a" ;
ATree = omitted,
Result = "t3b" ;
ATree = omitted,
Result = "t1" ;
ATree = omitted,
Result = "t3c" ;
ATree = omitted,
Result = "t2b" ;
ATree = omitted,
Result = "t3d" ;
false.
```

Note that `Result` unifies with a sequence of values in order. For brevity, the above output omits the printing of the value of `ATree`, which does not change.

Hint: the order that you write your predicate's clauses is significant.

2 Cut

In class, we discussed a simple predicate that will determine if an element is a member of a list:

```
ismember(H, [H|_]).
ismember(H, [_|T]) :- ismember(H,T).
```

We can use it to check membership like this:

```
?- ismember(1, [1,2,3,1]).
true ;
true ;
false.
```

Note, however, that because the queried element is present twice in the list, we get two `true` results. After the predicate succeeds the first time, Prolog will use *backtracking* and will continue searching until all possibilities have been exhausted.

Sometimes that might be desirable, but sometimes we only want to know if an item appear in a list at least once, making the extra backtracking unnecessary and wasteful.

The `!/0` predicate (pronounced *cut*) can be used to control backtracking. When this predicate is called, further backtracking terminates in the current predicate. We can therefore propose a modified predicate `ismember2`, that stops backtracking when the first matching element is found:

```
ismember2(H, [H|_]) :- !.
ismember2(H, [_|T]) :- ismember2(H,T).
```

When we use the new version of the predicate to check membership of a list, it returns at most one result:

```
?- ismember2(1, [1,2,3,1]).
true.
```

However, a disadvantage of using cut is that we can no longer use the predicate to enumerate all items in a list, because cut halts backtracking after the first result is found:

```
?- ismember2(X, [1,2,3,1]).
X = 1.
```

In another words, cut prevents the predicate from working in a backwards direction. This limits the usefulness of our predicate, and therefore we try to avoid using cut except when necessary.

Notice that cut can also change the logical meaning of your program. In the below example, cut prevents Prolog from considering other clauses of the same predicate. The logical meaning of each predicate is given on the right side in boolean algebra notation. Note that the two versions of `p` will yield different results when `a:-true`, `b:-false`, and `c:-true`.

`p :- a,b.` `p :- c.`	$p \Leftrightarrow (a \wedge b) \vee c$
`p :- a,!,b.` `p :- c.`	$p \Leftrightarrow (a \wedge b) \vee (\neg a \wedge c)$

With the help of cut, we are now in a position to understand the workings of the built-in `not/1` predicate. Here is a predicate `mynot`, which provides equivalent functionality.

```
mynot(A) :- A, !, fail.
mynot(_).
```

If the cut were not present, then `mynot` would backtrack so that it always succeeds (which is wrong). Make sure you understand how `mynot` works.

We can now write code like this:

```
?- between(1,5,X), mynot(X=3).
X = 1 ;
X = 2 ;
X = 4 ;
X = 5.

?- between(1,8,X), mynot(between(3,6,X)).
X = 1 ;
X = 2 ;
X = 7 ;
X = 8.
```

4. Write a predicate `sublist/2` such that `sublist(A,B)` succeeds for each matching sublist `A` contained in `B`, i.e. the sequence of terms in `A` appears as a contiguous sequence within `B`. Do *not* use cut, `\=`, `not`, or `\+`.

   For example, `sublist([a,a],[b,a,a,b])` succeeds, but `sublist([a,b,a],[b,a,a,b])` and `sublist([b,a,b],[b,a,a,b])` will fail. The call `sublist([a],[b,a,a,b])` will succeed *twice*. The call `sublist([a,b,d], [a,b,c,d])` will fail.

   Hint: it may be helpful to write a helper predicate.

5. Your implementation of `sublist/2` produces spurious results. For example, `sublist([a],[b,a,a,b])` will succeed twice. Use cut to write a modified version, `sublist_cut/2`, that will succeed at most once.

6. Without using your computer, give all the answers of `X` for each of the following queries:

   (a)
   ```
   member(X, [4,5, 14,15, 24,25]), X > 10, !, 0 is X mod 2.
   ```

   (b)
   ```
   member(X, [4,5, 14,15, 24,25]), X > 10, 0 is X mod 2.
   ```

   (c)
   ```
   member(X, [4,5, 14,15, 24,25]), !, X > 10, 0 is X mod 2.
   ```

   (d)
   ```
   member(X, [3,4,5, 13,14,15, 23,24,25]), X > 10, !, 0 is X mod
      ↪ 2.
   ```

7. One of the least efficient ways to sort a list is to generate all possible permutations of the elements in that list, and check which of those permutations are in sorted order. Fortunately(?), Prolog makes this sorting algorithm really easy.

   Write a predicate `bad_sort/2` such that `bad_sort(X,Y)` will succeed when list `Y` contains all the elements of list `X` in sorted, non-descending order, using the algorithm described above. Use the built-in `permutation/2` predicate to generate all possible permutations of the input list. Use cut to prevent spurious results. You may write a helper predicate.

   ```
   ?- bad_sort([5,3,1,10,3],Y).
   Y = [1, 3, 3, 5, 10].
   ```

   Hint: weirdly, the less-than-or-equal-to operator in Prolog is `=<`, rather than `<=`.

8. Write a predicate `lookup/3` that, given a list of `Key-Value` pairs, will succeed by giving the value corresponding to a key, or the key corresponding to a value.

   In effect, this predicate works analogously to a Python `dict` or a C++ `map`, except that it also allows reverse lookups.

   Use the cut operator to ensure that at most one result is returned.

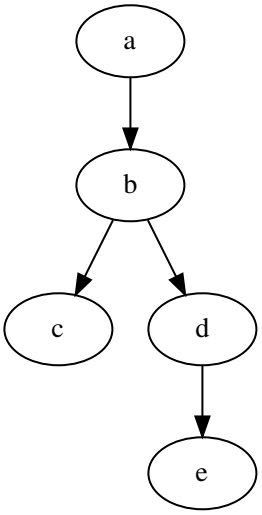
   ```
   ?- lookup(5, [6-a, 7-z, 5-t, 34-w], Value).
   Value = t.

   ?- lookup(9, [6-a, 7-z, 5-t, 34-w], Value).
   false.

   ?- lookup(Key, [6-a, 7-z, 5-t, 34-w], z).
   Key = 7.

   ?- lookup(6, [6-a, 7-z, 5-t, 34-w, 6-foo], Value).
   Value = a.
   ```

9. We will represent directed graphs in Prolog as a list of edges, such that each directed edge has the form `From-To`. That is, each edge consists of an atom representing the source node, and an atom representing the destination node, separated by a hyphen. For example, the graph `[a-b, b-c, b-d, d-e]` can be shown graphically like this:



Consider the following three graphs:

```
graph1([n1-n2, n2-n5, n1-n3, n1-n4, n4-n6, n6-n7, n6-n8]).

graph2([n1-n2, n2-n5, n1-n3, n1-n4, n4-n6, n6-n7, n7-n1, n7-n8]).

graph3([n4-n5, n1-n2, n1-n3, n1-n4, n4-n9, n9-10, n9-n11, n9-n12,
   ↪ n12-n9]).
```

The graph `graph1` is acyclic, while `graph2` clearly has a cycle, as does `graph3`. Enter the definition of `graph1`, `graph2`, and `graph3`, into your code file.

Write a predicate `hasCycles` such that the query `hasCycles(G)` will succeed when graph `G` contains a cycle, and will fail when `G` is an acyclic graph. For example:

```
?- graph1(G), hasCycle(G).
false.

?- graph2(G), hasCycle(G).
G = [n1-n2, n2-n5, n1-n3, n1-n4, n4-n6, n6-n7, n7-n1, n7-n8].
```

You may not make assumptions about the size or arity of the graph: your predicate must work with any graph.

Use the cut operator to ensure that your predicate will succeed at most once.

Hint: you will probably need to write a helper predicate.

Hint: you can use `member/2` to query elements in the graph representation. For example, the following query will unify `X` with all nodes that have an incoming edge from node `b`.

```
?- G=[a-b, b-c, b-d], member(b-X, G).
G = [a-b, b-c, b-d],
X = c ;
G = [a-b, b-c, b-d],
X = d.
```

3 Maze

Download the file `maze.pl`. It contains starter code for solving simple mazes. Load the file into Prolog and run the predicate `printUnsolvedMaze/0`, which will display the maze on the screen. The start position is indicated with an `S`, and the goal, or ending position, is indicated with an `E`. Asterisks mark walls, and spaces mark paths. Your task is to complete the program so that it will recursively search the maze to find a path from the start position to the ending position.

If you complete the program correctly, it will display the winning path on the screen graphically when you call `printSolvedMaze/0`, and the `winningPath/1` predicate will unify its parameter with a list of step-by-step directions from the starting position to the ending position.

Examine the code file. It's well commented, so read it carefully.

We use a (r, c) coordinate system, so each pair of coordinates indicates first the row number, then the column number.

Modify the following predicates as indicated. Do not modify any other code.

10. Complete the `newPos/5` predicate. It will output the new (r, c) position achieved after a move from the given position in one of the four cardinal directions: north, south, east, west. If, however, the proposed move would lead into a wall or off the maze board, the predicate will fail.

    Here we show how `newPos` can be used to evaluate the possible moves from position $(11, 1)$. See the listing below:

    ```
    ?- newPos(11,1,n,NewR,NewC).
    NewR = 10,
    NewC = 1 ;
    false.

    ?- newPos(11,1,w,NewR,NewC).
    false.

    ?- newPos(11,1,e,NewR,NewC).
    NewR = 11,
    NewC = 2 ;
    false.

    ?- newPos(11,1,s,NewR,NewC).
    NewR = 12,
    NewC = 1.
    ```

    As you can see, `newPos` can consider all four cardinal directions (north, south, west, east) but succeeds for moves only to the north, south, and east. The reason is that a move to the west would lead to

position $(11, 0)$, which is occupied by a wall according to `mazeWall/2`. The remaining directions lead us to positions $(10, 1)$, $(12, 1)$, and $(11, 2)$.

Hints:

- To avoid moving off the board, use predicate `mazeDimension/2`.
- To avoid moving into a wall, use predicate `mazeWall/2`.

11. Complete the `move/8` predicate. It will generate all moves from the current position, excluding moves that would move into a wall or off the board (`newPos` already takes care of that), and in addition excluding moves that would take us to a position where we've already been. To prevent infinite loops, we maintain a list of already-visited positions.

    The parameters of the predicate are as follows:

```
move(CurrentR, CurrentC,
   NewR, NewC,
   PosVisitedIn, PosVisitedOut,
   MoveListIn, MoveListOut)
```

    (a) `CurrentR,CurrentC` — The current row and column position.

    (b) `NewR,NewC` — A row and column position that we can move to.

    (c) `PosVisitedIn` — A list of `[R,C]` positions that have already been visited. We need to keep track of positions where we've already been to avoid retracing our steps and entering an infinite loop. The predicate may not produce a result found in this list.

    (d) `PosVisitedOut` — An amended list of `[R,C]` positions. The same as `PosVisitedIn`, with the new position added at the beginning.

    (e) `MoveListIn` — A list of cardinal directions that we've taken so far.

    (f) `MoveListOut` — Same as `MoveListIn`, with the new direction added at the beginning. That way, if we reverse this list, we can get a step-by-step guide to completing the maze. See `winningPath/1`.

    For example, here we use `move` to find all possible moves from the position $(11, 1)$.

```
?- move(11,1,NewR,NewC,[[11,2]],Visited,[w,w,w],Dirs).
NewR = 10,
NewC = 1,
Visited = [[10, 1], [11, 2]],
Dirs = [n, w, w, w] ;
NewR = 12,
NewC = 1,
Visited = [[12, 1], [11, 2]],
Dirs = [s, w, w, w] ;
false.
```

    As you can see, `move` considers all the moves given by `newPos`, but we exclude the possibility of moving to the east, because that would take us to an already-visited position. Therefore, only two directions are legal moves: north and south.

    In all cases, the predicate's result includes an updated list of visited positions and directions.

    You must call your `newPos` predicate from within `move`.

    Hints:

- Use `direction/1` to generate all possible cardinal directions. Feed its result to `newPos/5`, and then remove impossible results.

12. Complete the `solve/8` predicate. Its parameters are similar to those of `move/8`. The `solve/8` predicate should be recursive, repeatedly calling `move/8` to generate all possible paths through the maze. The base case occurs, and recursion stops, when a path to the maze's end position is found. At that time, the predicate's `PosVisitedOut` parameter should contain a list of all positions connecting the starting position to the ending position; and `MoveListOut` should contain step-by-step directions from the starting position to the ending position.

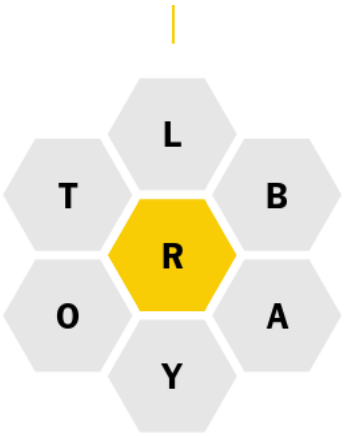
    Check your results by calling `printSolvedMaze/0` and `winningPath/1`. Note that Prolog will truncate the display of long solutions, so to see the complete winning path, use the query `winningPath(Path), writeln(Path)`.

    Submit your completed `maze.pl`.

4 Word finding

13. The New York Times' Spelling Bee puzzle gives you seven letters and requires you come up with as many English words as you can, such that each word is constructed only from those letters, with the additional requirement that each word must have at least four letters, and that each word must contain the central given letter.

    For example, consider the following puzzle:



    In this case, possible solutions would be "tray," "rotor," or "laboratory." However, "yabor" is not a solution, because it's not an English word. Also, "blot" is not a solution because it doesn't contain the central letter, "r," and "rat" is not a solution because it doesn't contain enough letters. Finally, "lottery" is not a solution because it contains the letter "e," which is not allowed. Note that correct solutions may use the same letter multiple times, and need not use all the given letters.

    Write a Prolog predicate `doSpellingBee/2` such that the query `doSpellingBee(Letters,Words)` with a list of letters in `Letters`, will answer with all matching words in `Words`. Matching words must be built only from the letters in `Letters`, and each matching words must contain the first letter in `Letters`.

    For example:

```
?- doSpellingBee([r,a,b,o,t,l,y],X).
X = [a, b, o, r, t] ;
X = [a, l, t, a, r] ;
X = [a, o, r, t, a] ;
X = [a, r, b, o, r] ;
X = [a, r, r, a, y] ;
X = [a, r, r, o, y, o] ;
X = [a, r, t, y] ;
etc...

?- doSpellingBee([t,e,v,o,q,m,n],X).
X = [e, m, o, t, e] ;
X = [e, n, t, e, n, t, e] ;
X = [e, v, e, n, t] ;
X = [m, e, e, t] ;
X = [m, e, m, e, n, t, o] ;
X = [m, e, t, e]
etc...
```

You have been given starter code in `spellingbee.pl`, as well as a text file containing a dictionary of English words, `spellingbee.txt`. In particular, please note the following predicates you have been given:

- `readDict(Dictionary)` will unify `Dictionary` with a list of words in the dictionary file, which must be in the same directory where you are running your program. Each word is expressed as a list of atoms. You should use this predicate in determining which combinations of letters are valid English words.

  ```
  ?- readDict(AllWords), member(SomeWord, AllWords),
     ↪ write(SomeWord), nl, fail.
  [a]
  [a,a,r,d,v,a,r,k]
  [a,a,r,d,v,a,r,k,s]
  [a,b,a,c,i]
  [a,b,a,c,k]
  [a,b,a,c,u,s]
  [a,b,a,c,u,s,e,s]
  [a,b,a,f,t]
  etc...
  ```

- `test1/0` and `test2/0` will help you test the correctness of your `doSpellingBee/2` predicate. For example:

  ```
  ?- test1.
  abort
  altar
  aorta
  arbor
  array
  arroyo
  arty
  ```

```
attar
barb
boar
boor
brat
...
```

- The other predicates in the file are helpers for `readDict/1` and will not help you directly.

Hint: you will probably want to write a helper predicate.

Hint: `length/2`, `member/2`, and `memberchk/2` will be helpful.

Add your implementation of `doSpellingBee/2` to the provided code file. Submit your completed `spellingbee.pl`.