Homework 3

Due Wednesday, Apr 10, 2024 at 8pm ET

The learning objectives of this homework are to:

- Understand how to write code in the logic-programming paradigm
- Gain experience specifically with writing code in Prolog

Use the following commands to download and unpack the distribution code:

```
$ wget https://eecs390.github.io/homework/hw3/starter-files.tar.gz
$ tar xzf starter-files.tar.gz
```

You may work alone or with a partner. Please see the syllabus for partnership rules. As a reminder, you may not share any part of your solution outside of your partnership. This includes code, test cases, and written solutions.

This assignment requires you to write code in Prolog. The officially supported interpreter for this course is SWI-Prolog. Installation instructions are on the download page. If you are using Mac OS, you can install with Macports or Homebrew, which will place the `swipl` interpreter in your path. You may also use the web-based version for this assignment. The CAEN machines have installations of SWI-Prolog:

```
module load swiprolog
```

You may find the built-in `length` and `append` predicates useful for this assignment.
You may **not** use the any of the following in this assignment:

- cuts (`!`) or any predicates that use cuts (e.g. `memberchk`, `->`)
- disjunction (`;`)
- the `clpfd` library

Exercises

1. *List manipulation.* Write a predicate `add_to_all` that, given three arguments, is true if the third is the result of inserting the first argument at the front of each list contained in the second argument. For example,

   ```
   add_to_all(a, [[], [b], [c, d]], [[a], [a, b], [a, c, d]])
   ```

   is true, whereas

   ```
   add_to_all(a, [[], [b], [c, d]], [[a, b], [a, c, d]])
   ```

   is not.

   Write your solution in `add_to_all.pl`.

   There are several tests in `add_to_all.in`. Each query ends in `!`, which is the Prolog *cut* operator, preventing more solutions from being explored once a solution has been found. We include this to ensure consistent output for different solutions to a problem.

   If you have SWI-Prolog installed, you can compare the results to the expected output using:

   ```
   $ swipl -q add_to_all.pl < add_to_all.in | diff - add_to_all.correct
   ```

2. *Merge sort.* In this question, we will implement merge sort on lists of integers. Write your code in `mergesort.pl`.

   a) Implement a `merge` predicate that relates two sorted lists to their merge:

```
?- merge([1, 3, 8], [2, 4, 5], Z).
Z = [1, 2, 3, 4, 5, 8] .
```

You will need to use the built-in comparison operations.

b) Now implement a `mergesort` predicate that relates a list to its sorted counterpart. Print out the "input" list in the recursive rule using `writeln`, as in the distribution code:

```
?- mergesort([4, 8, 5, 3, 1, 2, 6, 9, 7], X).
[4,8,5,3,1,2,6,9,7]
[4,8,5,3,1]
[4,8,5]
[4,8]
[3,1]
[2,6,9,7]
[2,6]
[9,7]
X = [1, 2, 3, 4, 5, 6, 7, 8, 9] .
```

When splitting up a list for the recursive terms, the two sublists should either be the exact same size, or the first sublist should be exactly one item larger than the second. You may find the built-in `length` and `append` predicates helpful here.

The tests above are in `merge.in` and `sort.in`.

3. *Matching.* In this question, we will implement an algorithm for matching people to a set of tasks. Each person expresses ranked preferences for some subset of the tasks, and the goal is to find an assignments of tasks to people such that everyone is assigned a task in their preference list. The algorithm will also take into account the preference ranks themselves when assigning a task.

   Consider a set of tasks, a set of people who can be assigned a task, and a set of preferences for each person for some subset of the tasks. The following is an example:

	Fe	**Wei**	**Gwyn**	**Ting**
Task 1	1		1	
Task 2	2			
Task 3		1	2	2
Task 4		2		1

   Here there are four tasks and four people, each with their top two preferences for tasks. Our goal is to determine a matching of tasks to people that takes into account each person's preference.

   We will implement a `match` predicate that can be used as follows:

```
?- match([[task1, [fe, 1], [gwyn, 1]],
          [task2, [fe, 2]],
          [task3, [wei, 1], [gwyn, 2], [ting, 2]],
          [task4, [wei, 2], [ting, 1]]
         ],
         [fe, wei, gwyn, ting], A).
```

   The first argument encodes the preferences. It is a list with one element per task, and a task itself is represented as a list consisting of the task name followed by preferences for that task. Each preference is a list containing the name of a person and their preference rank for the task. Thus, Task 1 is represented as `[task1, [fe, 1], [gwyn, 1]]`, where Fe ranks it as her first choice and Gwyn also ranks the task as her first choice.

   The second argument is a list of names of people who can be assigned tasks. The last argument is the actual resulting assignments of tasks to people. The following is one possible solution to the query above:

```
A = [[task1, gwyn, 1], [task2, fe, 2], [task3, wei, 1], [task4, ting, 1]].
```

   The overall algorithm we implement is as follows:

   1. Find the task with the fewest number of preferences. Prefer the leftmost if multiple tasks have the same number of preferences.

2. Assign a person to the task, out of the remaining unassigned people.
3. Remove the task from the list of remaining tasks.
4. Remove the assigned person from the set of remaining people.
5. Match the remaining tasks to the remaining set of people.
6. Add the assignment for the current task to the resulting assignment list from the previous step.

The beauty of logic programming is that if the algorithm gets stuck, e.g. by assigning a person to a task such that the remaining assignments are infeasible, it automatically backtracks to look for another solution.

We will start by implementing helper predicates before proceeding to `match`. Place all code for this question in `match.pl`.

a) Write a predicate `smaller` that is true if the first argument has a shorter length than the second argument:

```
?- smaller([1, 2], [3, 4, 5]).
true.

?- smaller([1, 2, 3], [3, 4, 5]).
false.
```

b) Now complete the implementation of the `smallest` predicate that relates a list of lists to the shortest list contained within the list of lists. If two lists have the same length, the leftmost such list should be preferred:

```
?- smallest([[1, 2], [3], [4], [5], [6, 7]], L), !.
L = [3].
```

In order for the matching algorithm to be efficient, the `smallest` predicate needs to be implemented tail recursively. As such, we write the `smallest_helper` predicate that relates a list of remaining items to the smallest item seen so far and the smallest item overall. Implement the recursive cases for this predicate.

You will need to prevent the interpreter from finding a solution that is not the leftmost of the shortest length, such as by using negation. The built-in `findall` predicate finds all solutions to a query. Your implementation of `smallest` should result in the following:

```
?- findall(L, smallest([[1, 2], [3], [4], [5], [6, 7]], L), Results).
Results = [[3]].
```

c) Implement an `assign` predicate that relates a list of preferences, a list of people, a person out of the latter list, and their preference rank from the first list. The elements of the first list are each a list of two atoms, the first being a person and the second their preference rank for the task.

The following is an example of a query:

```
?- assign([[fe, 3], [wei, 2], [gwyn, 3], [ting, 1]],
          [fe, gwyn, wei], Person, Rank), !.
Person = wei,
Rank = 2.
```

Your implementation should find the person with the minimum preference rank, as long as they are in the list of people in the second argument. However, it should not cut off the search space when it finds the minimal solution. Instead, if another solution is requested, it should find the person with the next lowest rank. Where two people have the same rank, it should first find the leftmost of the two in the preference list:

```
?- assign([[fe, 3], [wei, 2], [gwyn, 3], [ting, 1]],
          [fe, gwyn, wei], Person, Rank).
Person = wei,
Rank = 2 ;
Person = fe,
Rank = 3 ;
Person = gwyn,
Rank = 3 ;
false.
```

You will find the built-in `member` predicate useful, as well as one of the `sort` predicates. Find the documentation on the SWI-Prolog website.

With `findall`, your implementation of `assign` should result in the following:

```
?- findall([Person, Rank],
           assign([[fe, 3], [wei, 2], [gwyn, 3], [ting, 1]],
                  [fe, gwyn, wei], Person, Rank),
           Results).
Results = [[wei, 2], [fe, 3], [gwyn, 3]].
```

d) Finally, implement the `match` predicate, which relates a list of tasks, a list of people, and a list of assignments.

A task is a list where the first item is the task ID, and the remaining items are preferences for the task, each consisting of a list of a person and a preference rank:

```
[task1, [fe, 1], [gwyn, 1]]   % task1 is the task ID
```

The assignment list contains a list for each task, containing the task ID, the person assigned to it, and the preference rank of the person for the task:

```
[[task1, gwyn, 1], [task2, fe, 2], [task3, wei, 1], [task4, ting, 1]]
```

Implement the algorithm described above for `match`. You may find the built-in `select/3` predicate useful for Steps 3 and 4.

The following is an example, using the provided `sorted_match` predicate to produce a sorted list of assignments:

```
?- sorted_match([[task1, [fe, 1], [gwyn, 1]], [task2, [fe, 2]],
                 [task3, [wei, 1], [gwyn, 2], [ting, 2]],
                 [task4, [wei, 2], [ting, 1]]],
                [fe, wei, gwyn, ting], A), !.
A = [[task1, gwyn, 1], [task2, fe, 2], [task3, wei, 1], [task4, ting, 1]].
```

A more complex query, using the actual preferences for EECS 280 staff for lab sections from Winter 2017 (but with names changed), is in `match2.in`. Your implementation of `match` should be efficient enough to find a solution in less than a second.

e) *(Optional)* Write a new predicate `optimal_match` that finds an optimal matching, such that the sum total of the assigned preferences is minimized. As an example, the complex query in `match2.in` has an optimal assignment that results in a total of 58 rather than the 63 produced by the algorithm above.

Hint: The interpreter will continue to look for solutions after the first one. Come up with a way to force it to look for a better solution until no better solution exists. You will also need to cut off a search path as soon as it becomes clear that it cannot lead to a better solution.

Using our implementation, finding an optimal assignment takes about ten seconds on an iMac computer.

Submission

Place your solutions to question 1 in the provided `add_to_all.pl` file, to question 2 in the `mergesort.pl` file, and to question 3 in `match.pl`. Submit `add_to_all.pl`, `mergesort.pl`, and `match.pl` to the autograder before the deadline. Be sure to register your partnership on the autograder if you are working with a partner.