Fall 2021
Programming Languages
Homework 4

- This homework is a combination programming and "paper & pencil" assignment.
- Due via Brightspace on Tuesday, Dec 7, 2021 11:55 PM Eastern Time. Due to timing considerations relating to the end of the semester, late submissions will not be accepted. **No exceptions.**
- For the Prolog questions, you should use SWI Prolog.
- You may collaborate and use any reference materials necessary to the extent required to understand the material. All solutions must be your own, except that you may rely upon and use Prolog code from the lecture if you wish. Homework submissions containing any answers or code copied from any other source, in part or in whole, will receive a zero score.
- Please submit your homework as a zip file, `hw4-<netid>.zip`. The zip file should contain:
  - For Q1, `hw4-<netid>-Q1.pl` which should contain the reordered facts for question 1 and the 2 new rules for sub-question 4.
  - For the Prolog Rules question (Q2), `hw4-<netid>-prules.pl` which should contain your implementation of all of the rules. You do not need to submit queries, although we will run our queries on your solutions.
  - For Q3, a PDF file `hw4-<netid>-sols.pdf` containing the answers to all of the "paper & pencil" questions.
  - For Q4, `hw4-<netid>-escape.pl` which should contain all *rules* necessary to make the solution work. Use comments to mark your modifications.
  - For the Java programming question (Q5), a folder `hw4-prod_cons-<netid>`.
- Make sure your Prolog code compiles before submitting. *If your code does not compile for any reason, it may not be graded.*

1. [10 points] **Investigating Prolog**

   Consider the following:

```
male(brian).
male(kevin).
male(zhane).
male(fred).
male(jake).
male(bob).
male(stephen).
male(tom).
male(paul).

parent(melissa,brian).
parent(mary,sarah).
parent(stephen,jennifer).
parent(bob,jane).
parent(paul,kevin).
parent(tom,mary).
parent(jake,bob).
parent(zhane,melissa).
parent(tom,stephen).
parent(stephen,paul).
parent(emily,bob).
parent(zhane,mary).

grandfather(X,Y) :- male(X), parent(X,Z),parent(Z,Y).
```

   1. Reorder the facts above to provide faster execution time when querying `grandfather(tom,jennifer).` List the re-ordered facts and briefly explain what you changed.

      Answer:
      I placed parent(tom,stephen) before parent(tom,mary).

   2. Explain in your own words why the change above affects total execution time. Show evidence of the faster execution time (provide a trace for each).

      Answer:
      Because if I put parent(tom,stephen) before parent(tom,mary), prolog will not backtrace parent(tom,mary), which will reduce execution time.

      For original order, the trace is following:
      [trace] ?- grandfather(tom, jennifer).
      Call: (10) grandfather(tom, jennifer) ? creep
      Call: (11) male(tom) ? creep
      Exit: (11) male(tom) ? creep
      Call: (11) parent(tom, _19570) ? creep
      Exit: (11) parent(tom, mary) ? creep
      Call: (11) parent(mary, jennifer) ? creep
      Fail: (11) parent(mary, jennifer) ? creep

Redo: (11) parent(tom, _19570) ? creep
Exit: (11) parent(tom, stephen) ? creep
Call: (11) parent(stephen, jennifer) ? creep
Exit: (11) parent(stephen, jennifer) ? creep
Exit: (10) grandfather(tom, jennifer) ? creep
true

For my order, the trace is:
[trace] ?- grandfather(tom, jennifer).
Call: (10) grandfather(tom, jennifer) ? creep
Call: (11) male(tom) ? creep
Exit: (11) male(tom) ? creep
Call: (11) parent(tom, _14354) ? creep
Exit: (11) parent(tom, stephen) ? creep
Call: (11) parent(stephen, jennifer) ? creep
Exit: (11) parent(stephen, jennifer) ? creep
Exit: (10) grandfather(tom, jennifer) ? creep
true

So the trace is less than the original one, and the execution time is shorter.

3. Can we define a new rule `grandmother` that calls into the goals presented above to correctly arrive at an answer? Why or why not?

   Answer:
   Yes we can, because we can use `\+ male` to present female (under most cases).

4. Define new rules `aunt` and `uncle` that work with the rules above. If you need to define other rules (including facts) in order to correctly define these, go ahead. Assume that the universe of facts is **not** complete.

2. [30 points] **Prolog Rules**

Write the Prolog rules described below in a file `hw4-<netid>-prules.pl`. All rules must be written using the subset of the Prolog language discussed in class. You may not, for example, call built-in library rules unless specifically covered in the slides. You may call your own rules while formulating other rules. You do not need to turn in queries, although you should certainly test your rules using your own queries.

1. Write a rule `remove_items(I,L,O)` in which O is the output list obtained by removing every occurrence of the elements contained in List I from list L. The items in O should appear in the same order as L, only without the elements present in I.

```
Example:
?- remove_items([1,3],[2,6,7,7,8,3,1,1,4,3],O).
O = [2,6,7,7,8,4]
```

2. Write a rule `intersection2(L1,L2,F)`[1] in which F is a `set` containing only items that appear in both L1 and L2. You should not assume that L1 or L2 are sets[2] and, therefore, may contain duplicate items. Optional hint: first defining `intersection/4` and then define `intersection/3` in terms of that.

```
Example:
?- intersection2([2,3,4,3,5,8,2],[9,3,2,4,4],O).
O = [2,3,4]
```

3. Write a rule `disjunct_union(L1,L2,U)` in which U is the disjunctive union of the items in L1 and L2. That is, items in L1 or L2 that are not in the intersection of L1 and L2. You should not assume that L1 or L2 are sets.

```
Example:
?- disjunct_union([2,3,4,3,5,8,2],[9,3,2,4,4],O).
O = [5,8,9]
```

4. Write a rule `my_flatten(L1,L2)` which transforms the list L1, possibly holding lists as elements into a "flat" list L2 by replacing each list with its elements.

```
Example:
?- my_flatten([a, [b, [c, d], e]], X).
X = [a, b, c, d, e]
```

5. Write a rule `compress(L1,L2)` where all the repeated consecutive elements of the list L1 should be replaced with a single copy of the element. The order of the elements should not be changed.

```
Example:
?- compress([a,a,a,a,b,c,c,a,a,d,e,e,e,e],X).
X = [a,b,c,a,d,e]
```

6. Write a rule `encode(L1,L2)` where all the repeated consecutive duplicates of elements are encoded as terms [N,E] where N is the number of duplicates of the element E. The order of the elements should not be changed.

```
Example:
?- encode([a,a,a,a,b,c,c,a,a,d,e,e,e,e],X).
X = [[4,a],[1,b],[2,c],[2,a],[1,d][4,e]]
```

7. Modify the result of previous problem in such a way that if an element has no duplicates it is simply copied into the result list. Only elements with duplicates are transferred as [N,E] terms. Name the rule as `encode_modified`.

[1] The rule `intersection` is already defined by the Prolog standard library.
[2] A *set* is a collection of unique, unordered items.

```
Example:
?- encode_modified([a,a,a,a,b,c,c,a,a,d,e,e,e,e],X).
X = [[4,a],b,[2,c],[2,a],d,[4,e]]
```

8. Write a rule `rotate(L1,N,L2)` where the list L2 is the modified version of list L1, with the elements rotated as N places to the left if N is a positive number, otherwise rotated as N places to the right. You can assume that list L1 is never empty and $|N| < length(L1)$, where $|.|$ is the absolute value operator.

```
Examples:
?- rotate([a,b,c,d,e,f,g,h],3,X).
X = [d,e,f,g,h,a,b,c]

?- rotate([a,b,c,d,e,f,g,h],-2,X).
X = [g,h,a,b,c,d,e,f]
```

3. [10 points] **Unification**

For each expression below that unifies, show the bindings. For any expression that doesn't unify, explain why it doesn't. Assume that the unification operation is left-associative.

1. d(15) & c(15)

   It doesn't unify, because the functors are different.

2. 4 & X & 76

   It doesn't unify, because 4 and 76 cannot unify.

3. a(X, b(3, 1, Y)) & a(4, Y)

   It doesn't unify, because Y b(3, 1, Y) will create an infinite recursion.

4. b(1,X) & b(X,Y) & b(Y,1)

   It can unify, with X = Y = 1.

5. a(1,X) & b(X,Y) & a(Y,3)

   It doesn't unify, because the functors are different.

6. a(X, c(2, B, D)) & a(4, c(A, 7, C))

   It can unify, with X = 4, A = 2, B = 7, C = D.

7. e(c(2, D)) & e(c(8, D))

   It doesn't unify, because 2 and 8 cannot unify.

8. X & e(f(6, 2), g(8, 1))

   It can unify, with X = e(f(6, 2), g(8, 1)).

9. b(X, g(8, X)) & b(f(6, 2), g(8, f(6, 2)))

   It can unify, with X = f(6, 2).

10. a(1, b(X, Y)) & a(Y, b(2, c(6, Z), 10))

    It doesn't unify, because there are b/2 and b/3.

11. d(c(1, 2, 1)) & d(c(X, Y, X))

    It can unify, with X = 1, Y = 2.

4. [20 points] **Escape from the Vault**

Tokyo, Rio, Berlin and Denver have just looted the Museum of Modern Arts and need to cross the Kosciusko Bridge in order to reach a deserted runway where Professor Plock's aeroplane is waiting for them. However, the bridge is fragile and can hold at most two of them at the same time. Moreover, to cross the bridge a flashlight is needed to avoid traps and broken parts. The problem is that there is only one flashlight with one battery that lasts for only 60 minutes. Each of them need different times to cross the bridge (in either direction):

PERSON	TIME
Tokyo	5 minutes
Rio	10 minutes
Berlin	20 minutes
Denver	25 minutes

Since there can be only two people on the bridge at the same time, they cannot cross the bridge all at once. Since they need the flashlight to cross the bridge, whenever two have crossed the bridge, somebody has to go back and bring the flashlight to the people on the other side that still have to cross the bridge. The problem now is: In which order can the above four cross the bridge in time (that is, in 60 minutes) to be saved from getting caught? Being the mastermind of the heist, you need to find the answer to the above conundrum. You could have solved the problem in your head, had you not been day drinking. The only possible way for you now is to try all possible combinations. Curiously, the alcohol seems to have no effect on your ability to write logically correct software. So, you decide to write some Prolog rules to answer this.

Writing a Prolog program for solving the riddle is in principle a rather straightforward task (using backtracking) — at least once one has figured out how to formulate the solution. The most difficult part is to find an appropriate term representation for the states of the search problem, i.e., the position of people on either side of the bridge and the position of the flashlight. We are going to develop one such idea here. Let us assume that initially, everyone is on the left side of the bridge and want to cross over to the right side. We represent intermediate states of a bridge crossing by facts of the form `st(P,L)` where `L` is a list giving the people that are currently on the left side of the bridge and where `P` is a flag that indicates the position (left or right side) of the flashlight. We denote movement from left to right as the rule `r(L1)` where `L1` is a list of people who move from left to right. Similarly, we define `l(L2)` to denote movement from right to left.

1. Write a rule `time(P,T)` where `T` is the time taken by person `P` to cross the bridge (in either direction).
2. Write a fact `team(T)` where `T` is a list of all four people who need to cross the bridge.
3. Write a rule `cost(L,C)` where `C` is the maximum time required for all the people in list `L` to cross the bridge (in either direction), assuming that they all can cross the bridge simultaneously.
4. Write two rules `move(st(l,L1), st(r,L2), r(M), C)` and `move(st(r,L1), st(l,L2), l(M), C)`, where:
   - `move(st(l,L1), st(r,L2), r(M), C)`: This movement is generated if the flashlight is on the left side of the bridge. `C` is the time required to move people contained in the list `M` from left to right, where `st(l,L1)` is the old state and `st(r,L2)` is the new state. In this case, the time is determined by the predicate `cost/2`. The list M of people to move to the right is computed by the predicate `split/3`, which computes lists of length 2, which are sorted to avoid redundancy caused by representing groups of people as lists. Implementation of `split/3` is provided to you:

```
split(L,[X,Y],M) :-
    member(X,L),
    member(Y,L),
```

```
compare(<,X,Y),
subtract(L,[X,Y],M).
```

Here, `M` is the list after removing `X` and `Y` from the list `L`.

- `move(st(r,L1), st(l,L2), l(M), C)`: This movement is generated if the flashlight is on the right side of the bridge. `C` is the time required to move people contained in the list `M` from right to left on the bridge, where `st(r,L1)` is the old state and `st(l,L2)` is the new state. In this move, it makes sense to send back only one person. Therefore, the definition of move for this case uses the predefined `member/2` [3] predicate and computes the time by simply looking into the rule `time/2`.

5. Finally, the `trans/4` predicate generates all possible bridge crossings together with the required time, i.e., write a rule `trans(I,F,M,C)` where `C` is the amount of time needed to reach the final state `F` from initial state `I` and `M` is the ordered list of movements(refer to the example below for a clear understanding).

The `cross/2` predicate formulates the search problem by giving the initial and final configuration of the search space. The definition of `cross/2` is as follows:

```
cross(M,D) :-
    team(T),
    trans(st(l,T),st(r,[]),M,D0),
    D0=<D.
```

The solution to the original problem is then given by:

```
solution(M) :- cross(M,60).
```

There are two possible answers to the above problem. So, upon executing `solution/1` the final output should be:

```
?- solution(M).
M = [r([rio, tokyo]), l(tokyo), r([berlin, denver]), l(rio), r([rio, tokyo])] ;
M = [r([rio, tokyo]), l(rio), r([berlin, denver]), l(tokyo), r([rio, tokyo])] .
```

Note: The order of the above solutions does not matter.

[3] https://www.swi-prolog.org/pldoc/man?predicate=member/2

5. [20 points] **Producer-Consumer** [To be completed after the Nov 30 lecture]

The producer-consumer problem is well known in the context of multi-threaded programs. The problem is as follows: We have a buffer of fixed size. A producer can produce an item and can place in the buffer. A consumer can "consume" items (i.e., remove an item from the buffer and do something useful with it). This must be accomplished in such a way that none of the items get lost, duplicated, or corrupted during any run of the program, and the consuming must be carried out efficiently. The problem can be generalized to multiple producers and multiple consumers.

This problem gives rise to two issues:

- Because producing and consuming can occur simultaneously, a critical section is needed to protect the buffer such that either only one producer or only one consumer may access the buffer at any time.
- The consumer needs to know when to check the buffer. One highly discouraged method is to use a "busy-wait" loop that constantly checks the buffer for items and consumes them if any are found. The problem with this approach is that the loop will max out the CPU, thus considerably slowing down the other threads and is therefore highly inefficient. A better way is to use the **notify** and **wait** construct we learned in class.

Your task in this assignment is to create a multi-threaded Java program to do the following: the Server class should create two producer threads identified as "Alice" and "Moira", each having their own distinct buffer. There should be one consumer thread identified as "Johnny." Johnny must be notified whenever a new item is placed in either buffer and must also be notified which buffer to check. (Johnny may not guess which buffer to check or check both). Implement this logic using the **notify** and **wait** constructs. To be clear, Alice and Moira place items into their own separate buffers. Johnny consumes the items from the two buffers as items are placed in them. There is one very important twist to the buffers: each buffer may hold at most one item at a time.

Each producer thread repeats the following steps six (6) times:

1. sleep for a random amount of time (ranging from 1 to 3 seconds)
2. print the producer name and sleep time before going to sleep
3. generate a non-negative random integer value in the range 4000 to 60000
4. print the producer name and the generated value
5. deposit this value into the proper buffer.

The corresponding consumer thread repeats the following steps twelve (12) times:

1. pick up a value from the buffer
2. print the producer name and the value picked up

A few important points to note:

1. Each producer should produce a total of six numbers and place these six numbers in their respective buffers. Hence, the consumer should consume a total of twelve numbers.
2. As noted above, each buffer can only hold one item at a time. If the buffer is full, the producer seeking to deposit a new item must wait until its buffer becomes empty.
3. A busy-wait loop may not be used to check the status of the buffer. Also, do not write a busy-wait loop with a sleep statement inside it with the intent of "slowing down" the loop. Rather, you must use the **notify** and **wait** construct we learned in class to inform the consumer of the presence of a new item.

Following is the snapshot of one of the possible output. Your output should be similar to this (note that the generation of numbers and sleep time is non deterministic, so, your output might look a bit different from this. However, a consumer can consume only if there is something in the buffer.)

```
Producer Alice sleeping for 1 seconds
Producer Moira sleeping for 2 seconds
Producer Alice produced 35526
Producer Alice sleeping for 3 seconds
Consumer Johnny consumed 35526 produced by producer Alice
Producer Moira produced 56928
Consumer Johnny consumed 56928 produced by producer Moira
Producer Moira sleeping for 2 seconds
Producer Moira produced 48837
Producer Alice produced 6380
Producer Alice sleeping for 1 seconds
Consumer Johnny consumed 48837 produced by producer Moira
Producer Moira sleeping for 2 seconds
Consumer Johnny consumed 6380 produced by producer Alice
Producer Alice produced 25342
Producer Alice sleeping for 3 seconds
Consumer Johnny consumed 25342 produced by producer Alice
Producer Moira produced 38858
Producer Moira sleeping for 2 seconds
Consumer Johnny consumed 38858 produced by producer Moira
Producer Moira produced 16813
Producer Alice produced 12416
Producer Alice sleeping for 2 seconds
Consumer Johnny consumed 16813 produced by producer Moira
Producer Moira sleeping for 2 seconds
Consumer Johnny consumed 12416 produced by producer Alice
Producer Alice produced 8734
Producer Moira produced 47869
Producer Moira sleeping for 2 seconds
Consumer Johnny consumed 8734 produced by producer Alice
Producer Alice sleeping for 1 seconds
Consumer Johnny consumed 47869 produced by producer Moira
Producer Alice produced 34332
Consumer Johnny consumed 34332 produced by producer Alice
Producer Alice EXIT
Producer Moira produced 78560
Producer Moira EXIT
Consumer Johnny consumed 78560 produced by producer Moira
Consumer Johnny EXIT
```

You must implement the solution to the above problem in Java. For your convenience, this homework comes with sample source code for a producer/consumer problem based on shared buffer which you may edit/modify to do this question (See the zip file `prod_cons.zip`). Note that this folder includes the following files: `Server.java`, `Producer.java`, `Consumer.java`, and `BoundedBuffer.java`. You will rename the folder `prod_cons` replacing `prod\_cons` with `hw4-prod_cons-<netid>` and submit it as a solution for this question.

6. [10 points] **Prototype OOLs**

Consider the following code below, written in a prototype OOL:

```
var obj1;
obj1.x = 20;
var obj2 = clone(obj1);
var obj3 = clone(obj2);
var obj4 = clone(obj1);
obj2.y = 5;
obj4.x = 10;
obj3.z = 30;
```

Assume that the program fragment above has executed. Answer the following:

1. What fields are contained locally to `obj1`?

   x is contained locally to `obj1`

2. What fields are contained locally to `obj2`?

   x and y are contained locally to `obj2`

3. What fields are contained locally to `obj3`?

   x and y and z are contained locally to `obj3`

4. What fields are contained locally to `obj4`?

   x is contained locally to `obj4`

5. To what value would `obj1.x` evaluate, if any?

   obj1.x = 20

6. To what value would `obj2.x` evaluate, if any?

   obj2.x = 20

7. To what value would `obj3.x` evaluate, if any?

   obj3.x = 20

8. To what value would `obj4.x` evaluate, if any?

   obj4.x = 10

9. To what value would `obj4.y` evaluate, if any?

   obj4.y is not defined.

10. To what value would `obj2.y` evaluate, if any?

    obj2.y = 5

11. To what value would `obj3.y` evaluate, if any?

    obj3.y = 5

12. To what value would `obj3.z` evaluate, if any?

    obj3.z = 30