CPS721: Assignment 3

Due: October 24, 2023, 9pm
Total Marks: 100 (worth 4% of course mark)
You MUST work in groups of 2 or 3

Late Policy: The penalty for submitting even one minute late is 10%. Assignments are not accepted more than 24 hours late.

Clarifications and Questions: Please use the discussion forum on the D2L site to ask questions as they come up. These will be monitored regularly. Clarifications will be made there as needed. A Frequently Asked Questions Page will also be created. You may also email your questions to your instructor. Check the D2L forum and frequently asked questions first.

Collaboration Policy: You can only discuss this assignment with your group partners or with your CPS721 instructor. By submitting this assignment, you acknowledge that you have read and understood the course policy on collaboration as stated in the CPS721 course management form.

PROLOG Instructions: When you write your rules in PROLOG, you are not allowed to use ";" (disjunction), "!" (cut), and "->" (if-then). You are only allowed to use ";" to get additional responses when interacting with PROLOG from the command line. Note that this is equivalent to using the "More" button in the ECLiPSe GUI.

We will be using ECLiPSE Prolog release 6 to mark the assignments. If you run any other version of PROLOG, it is your responsibility to check that it also runs on this version.

SUBMISSION INSTRUCTIONS: You should submit ONE zip file called `assignment3.zip` containing 6 files:

`q1a_puzzle_generate_and_test.pl`	`q1b_puzzle_interleaving.pl`
`q1_puzzle_session.txt`	
`q2_network.pl`	`q2_nework_session.txt`
`q3_terms.pl`	`q3_terms_session.txt`
`(OPTIONAL) q4_food.pl`	`q4_food_session.txt`

These files have been given to you and you should follow the format given. Your submission should not include any other files. If you submit a `.rar`, `.tar`, `.7zip`, or other compression format aside from `.zip`, you will lose marks. All submissions should be made on D2L. Submissions by email will not be accepted. As long as you submit your assignment with the file name `assignment3.zip` your group will be able to submit multiple times as it will overwrite an earlier submission. You do not have to inform anyone if you do. The time stamp of the last submission will be used to determine the submission time. Do not submit multiple `zip` files with different names. If you do, we will use the last submitted one, but you may lose marks.

Make sure the files are saved in plain text and readable on Linux machines. Ensure your PROLOG code does not contain any extra binary symbols and that they can be compiled by ECLiPSE Prolog release 6.

1 Crypt-Arithmetic [30 marks]

Use Prolog to solve the following crypt-arithmetic puzzle involving **multiplication**:

```
          L  E  T
       *  L  A  P
-----------------
  +       I  T  S
    L  O  V  E  .
    L  E  T  .  .
----------------
    T  O  O  L  S
```

This notation means that LET * LAP = TOOLS, and the words between the bars correspond to performing the usual multiplication algorithm. That is ITS + 10*LOVE + 100*LET=TOOLS. For example, the calculation of 123 * 12 would appear as

```
          1  2  3
       *     1  2
-----------------
  +       2  4  6
       1  2  3  .
----------------
       1  4  7  6
```

Each letter stands for a distinct digit and leading digits are not zeroes. You cannot make any other assumptions. You will solve this problem in two parts.

a. [10 marks]

First, try to solve this problem using the *pure generate and test* technique, without any interleaving, and notice how much time it takes. In particular, you should write the rules for *solve*, which takes in your list of variables and finds an assignment that solves the puzzle. You should also write the rules for *solve_and_print* which calls your *solve* rule and prints out your solution in a human-readable form. These should be put in the given file `q1a_puzzle_generate_and_test.pl` following the directions given in the file. Ensure that both the *solve* and *solve_and_print* can be run.

You can then determine how much computer time your computation takes using the following query:

```
?- Start is cputime, solve_and_print, End is cputime, Time is End - Start.
```

The value of `Time` will be the time taken. Note, that the timing code should not be included in `q1a_puzzle_generate_and_test.pl`.

Add the results of this query, including the time and solution, to the file `q1_puzzle_session.txt` in the correct section.

b. [20 marks] Next, solve this problem using a smart interleaving of generate and test. Again implement *solve* and *solve_and_print*, and add them to the file `q1b_puzzle_interleaving.pl`. In this file, add comments in your file which explain briefly the order of constraints you have chosen and why this has an effect on computation time. You can draw a dependency graph by hand and include a PDF image if you decide to use one (but it is not a requirement).

Find also how much time your program takes to compute an answer in this case. Again, include the results of your session in `q1_puzzle_session.txt`

2 Logic Puzzle [40 marks]

In the town Mooseville, there are just six computer systems connected to the Internet, though each system may have many users on it. The six computer systems exchange files with the systems linked with them on the map below, and system 1 on the map also exchanges files with systems outside of Mooseville, providing the town with its connection to the rest of the world. Thus, electronic mail travels from system to system along the links shown on the map:

```
(6) -- (5) -- (3) -- (2) -- (1) --> (Rest of the world)
                |
               (4)
```

`Note: 4 has a link to 3, 5 has a link to 3, 5 does not have a direct link to 4. The path from 6 to "rest of the world" is: 6, 5, 3, 2, 1, out.`

Each of the computer systems has a different Internet address (`bananas.com`, `firstbank.com`, `netvue.com`, `pricenet.com`, `sysworld.com`, and `univmoose.edu`), and a different man or woman as its systems administrator. The six systems administrators include three women named Catarina, Lizzie, and Mona, and three men named Anthony, Daniel, and Jaime; last names are (not respectively) Elby, Kim, Osborne, Tsuji, Wolverton, and Zickerman. Your Prolog program must match each computer system on the map with its Internet address and with the full name of its system administrator. Suppose the following facts are true.

1. Email between Lizzie and Osborne must pass through the system with the Internet address `pricenet.com` (and possibly other systems as well).
2. Mona's and Wolverton's systems exchange files directly; one of these systems' addresses is `netvue.com`
3. Email between Anthony and Jaime must pass through Elby's system (and possibly others).
4. Among the systems in town, Daniel's exchanges files directly only with the one whose address is `sysworld.com`
5. Email between Jaime and Ms. Tsuji must also pass through the system whose address is `univmoose.edu`
6. The system whose address is `bananas.com` has a woman systems administrator.
7. Email between Kim and the rest of the world must also pass through the system whose address is
   `firstbank.com`
8. Among the systems in town, Zickerman's exchanges files directly only with Catarina's and the system whose address is `netvue.com`. Zickerman's system does not have direct links with any other computer system in the town.

Assume that in the constraints (1), (3), (5), (7) the packages can possibly pass through more than one system (not only one computer system that is mentioned explicitly).

a. **[10 marks]** Implement the 3-argument predicate *trip(Origin,Destination,Path)* as a helping predicate and use it in your program. This predicate is true, if *Path* is the list of computer systems connecting *Origin* with *Destination* (*Path* includes computer systems *Origin* and *Destination* if they are different from each other). If *Origin=Destination*, then *Path* has only

one element. There is a direct link on the map between consecutive elements of *Path*. Using this predicate *trip*, you can formulate the constraint that email from *Origin* to *Destination* must pass through a computer system *Intermediate* (in other words, that the computer system *Intermediate* is located between *Origin* and *Destination* in the computer network on the map).

b. [**25 marks**] Write a Prolog program `q2_network.pl` that solves this problem using interleaving of generate and test technique considered in class. You will need to be careful in your program regarding the order of constraints. You should implement both a *solve* and *solve_and_print* as described in question 1. Explain briefly the ordering you have chosen (write comments in you program file). Make sure that *solve_and_print* prints the answers returned by your program in an easy to read format: you will lose marks if output is obscure.

c. [**5 marks**] We should be able to query your program using both *solve* and *solve_and_print*. Make sure it is obvious from your output what the solution is. Find also how much time your program takes to compute an answer using the process described in the previous Part 1 of this assignment. Your Prolog session should be included in the file `q2_network_session.txt` Thus, your submission should include both `q2_network.pl` and `q2_network_session.txt`

3 Terms [30 marks]

This part will exercise what you have learned about recursion over terms in Prolog. Let term *next(Head,Tail)* represent a Prolog list $[Head \mid Tail]$, where *nil* represents the empty list `[]`. For example, *next(7, next(1, next(5, next(0, next(9,nil)))))* represents the list *[7,1,5,0,9]*.

All your programs should be added to the file `q3_terms.pl`. In addition, you should include sessions with your tests of the three programs in `q3_terms_session.txt`. In your sessions, you should request all answers (using either ";" or the button **more**)

a. [**5 marks**] Implement the predicate *appendT(Term1,Term2,Result)*. This predicate is similar to the predicate *append(List1,List2,Result)* that we discussed in class, but it works with terms representing lists. The following are examples of predicates that succeed.

```
?- appendT(next(a, next(b, nil)), next(1, next(2, nil)), Result).
        Result = next(a, next(b, next(1, next(2, nil))))
?- appendT(Init, next(1,next(2,nil)), next(a,next(b,next(1,next(2,nil))))).
        Init = next(a, next(b, nil))
?- appendT(next(a,next(b,nil)), Final, next(a,next(b,next(1,next(2,nil))))).
        Final = next(1, next(2, nil))
```

Add your rules to the file `q3_terms.pl` in the corresponding section.

b. [**10 marks**] Implement the predicate *list2term(List,Term)* that takes as its input a usual Prolog List, and produces as its output a Term representing this list. You can use the library predicates `var(X)` and `atom(X)`. Note that `var(X)` succeeds if `X` is a variable (*ie.* `var(A)` succeeds while `var(a)` fails,) while `atom(X)` succeeds if `X` is an atom (*ie.* `atom(a)` succeeds but `atom(X)` and `atom(term(a))` fails). Note that both *atom*([*nil*]) and *atom*([ ]) will succeed.

The following are examples of queries using `list2term` that succeed:

```
?- list2term([[a], [b, [c]] | [d] ], X).
     X= next(next(a,nil), next(next(b,next(next(c,nil),nil)), next(d,nil)))
?- list2term([a | [b | [c]] ], X).
     X= next(a, next(b, next(c, nil))).
?- list2term([ [a,[b, [c]]], d], X).
     X= next(next(a, next(next(b, next(next(c,nil), nil)), nil)), next(d,nil))
?- list2term([[a, b]], X).
     X= next(next(a, next(b, nil)), nil).
```

Add your rules to the file `q3_terms.pl` in the corresponding section.

c. [**15 marks**] Implement *flat(Term,FlatTerm)*. Here, *Term* is a given term representing list of lists (they can also contain nested lists inside), and *FlatTerm* is a *Term* 'flattened' so that the elements of *Term*'s sublists (or sub-sublists) are reorganized into a term representing one plain list (that has no sublists), but the sequence of elements remains the same. You can assume in your program that there are no occurrences of empty lists in *Term* (*ie.* `next([], nil)` is not allowed), but *Term* itself can be *nil* (i.e., the empty list). The order of elements occurring in *Term* remain the same in *FlatTerm*. There should be no terms inside *FlatTerm* that would represent nested lists. Examples of queries that succeed:

```
?- list2term([[a, b]], X), flat(X, Y).
        X = next(next(a, next(b, nil)), nil)    Y = next(a, next(b, nil))
?- list2term([[[a] | [z]],[[b,[c]],[d]]], Term), flat(Term,FlatTerm)
  Term= next(next(next(a,nil), next(z,nil)),
```

```
       next(next(next(b, next(next(c,nil),nil)), next(next(d,nil),nil)),nil))
  FlatTerm = next(a, next(z, next(b, next(c, next(d, nil)))))
?- list2term([[k],[[l]],[m | [n]]], Term), flat(Term, FlatTerm).
  Term= next(next(k,nil), next(next(next(l,nil),nil),
               next(next(m, next(n,nil)),nil)))
  FlatTerm = next(k, next(l, next(m, next(n,nil))))
```

Add your rules to the file `q3_terms.pl` in the corresponding section.

4 BONUS [30 marks]

> To make up for a grade on another assignment or test that was not what you had hoped for, or simply because you find this area of Artificial Intelligence interesting, you may choose to do extra work on this assignment. *Do not attempt any bonus work until the regular part of your assignment is complete.* If your assignment is submitted from a group, write whether this bonus question was implemented by all people in your team (in this case and by default bonus marks will be divided evenly between all students) or whether it was implemented by one student only (in this case only this student will get all bonus marks).

John ordered enough takeout food from a local Asian food restaurant to eat for dinner for 5 straight days, but he only got 5 different dishes: *egg rolls, chow mein, sour soup, fried rice* and *Peking duck*. John always ate two courses at every meal. To keep his meals interesting, he mixed the dishes up so that he never ate the same combination twice in one week. John did not have this problem with his drink order as he drank something different with every meal. He liked to drink *tea, coffee, milk, juice and water*. Write a Prolog program based on the following clues.

1. John never ate the same combination of dishes twice in one week.
2. The only dish John ate two days in a row was *Peking duck* on Tuesday and Wednsday.
3. John didn't order *sour soup* on Tuesday, Wednesday or Thursday.
4. John's fist course on Tuesday was *chow mein*, and he had *chow mein* again for second course the day after his 1st course was *duck*.
5. John ate no *egg rolls* on Monday or Thrusday, and he drank *coffee* after he drank *milk*, and *tea* the day after he drank *coffee*.
6. John ordered *egg rolls* at least twice in a week.
7. *Sour soup* can be ordered only as the first dish, but not as the second.
8. John drank *water* afer he drank *tea*, and he drank *juice* on Friday.

Your task is to write a PROLOG program to solve this problem using the smart interleaving of generate-and-test technique explained in class: your program has to compute the full meal on each weekday including two courses and a drink. Do not attempt to solve any part of this puzzle yourself, i.e., do not make any conclusions from the statements given to you. To get full marks, you have to follow a design technique from class.
Hints: introduce a predicate for weekdays. Using this first predicate, write atomic statements that define a finite domain for all variables related to drinks. Additionally, introduce another predicate for dishes and use it to write atomic statements. The variables for the first and second courses per weekday should take values from the domain of dishes.

You will have to be careful in your program regarding the order of constraints. Explain briefly the ordering you have chosen (write comments in your program). Remember to implement all implicitly stated and hidden constraints. They must be formulated and included in the program to find a correct solution satisfying all explicit and implicit constraints. You can use predicates from class in your program. Determine how much computer time your computation takes using `cputime` construct. Never try to guess part of solution by yourself: all reasoning should be done by your program. You have to demonstrate whether you learned well a program design technique. You lose marks, if the TA will find that you embed some of your own reasoning into your program.

You should implement both a *solve* and *solve_and_print* as described in question 1. These should be put in the file `q4_food.pl`. Explain briefly the ordering you have chosen (write comments in you program file). Make sure that *solve_and_print* prints the answers returned by your program in an easy to read format: you will lose marks if output is obscure. You should also include your session testing your program in `q4_food_session.txt`. This test should include information about the runtime. Add a brief description of your program in this file as well.