cps721: Assignment 3 (100 points).
Due date: Electronic file - Monday, October 26, 2020, 21:00 (sharp). Read Page 4.

You must work in groups of TWO, or THREE. You cannot work alone.

YOU SHOULD NOT USE ";" "!" AND "–>" IN YOUR PROLOG RULES.

> You can discuss this assignment only with your CPS721 group partners or with the CPS721 instructor. By submitting this assignment you acknowledge that you read and understood the course Policy on Collaboration in homework assignments stated in the CPS721 course management form.

This assignment will exercise what you have learned about constraint satisfaction problems (CSPs). In the following questions, you will be using Prolog to solve such problems, as we did in class. For each of the following questions below, you should create a separate file containing rules for the `solve(List)` predicate and any other helping predicates you might need. Note that it is *your program that should solve each of these problems*, not you! This means that no part of the solution can be hard-coded in your program. You lose marks, if you attempt to solve a problem (or any part of the problem) yourself, and then build-in your solution into your program. All work should be done by **your Prolog program**, not by you.

Note that in all parts of this assignment you have to implement a Prolog rule with the predicate `solve(List)` or similar in the head, and this predicate must have only one argument, the list of variables. In addition, your task is to implement the predicate `print_solution(List)` that has a list of variable as its argument, calls `solve(List)` and then formats the output to make it more readable. This uniformity is important since the TA will run a script to test your programs.

1 (35 points). Use Prolog to solve the following crypt-arithmetic puzzle involving subtraction:

```
  _ SANTA
    CLAUS
 --------
     XMAS
```

Assume that each letter stands for a distinct digit and that leading digits are not zeroes. Make sure that your output is easy to read: format your output using the predicate `write(X)` and `nl` (see examples in slides).

First, solve this problem using pure generate and test technique, without any smart interleaving, and notice how much time it takes. (Warning: it can take about a minute.) Determine how much computer time your program takes in the rule for `print_solution(List)` using the following pseudo-code that calls `cputime` (do not count printing time):

```
StartTime is cputime, <call solve(L)>,  EndTime is cputime, Time is EndTime - StartTime.
```

The value of `Time` will be the time taken (include this information in comments). Submit this program in the file **puzzle1.pl**

Next, solve this problem using smart interleaving of generate and test. Find also how much time your program takes to compute an answer. Write comments in your program file: **explain briefly** the order of constraints you have chosen and **why** this has an effect on computation time. If you decide to use a dependency graph, you can draw it by hand, but make sure it is legible. You can save your drawing in a PDF file and include it into your zip archive. Keep the second program in the file **puzzle2.pl** When a TA will be testing your program, he will call the predicate **print_solution(List)** in both programs. Make sure this query can be executed, and that it will output clearly a **readable solution** that your program found.

Handing in solutions: An electronic copy of your files **puzzle1.pl** and **puzzle2.pl** must be included in your **zip** archive.

2 (35 points). Use Prolog to solve the following problem related to computing a timetable for an imaginary department of computer science that has 4 class-rooms ($r1, r2, r3, r4$) available, and where classes can be potentially scheduled. Use the list with 3 arguments *[Day,Hour,Room]* to represent a class meeting in *Room* on *Day* at *Hour*. In the department, some dates and times are already taken by other courses. These taken times and class-rooms are represented by the following atomic statements (notation is similar to the example that we discussed in class; see also a posted PDF file with slides).

```
taken([mon,12,r1]). taken([mon,1,r1]). taken([mon,1,r4]). taken([mon,2,r4]).
taken([tues,9,r2]). taken([tues,10,r2]). taken([tues,3,r1]). taken([tues,4,r1]).
taken([wed,10,r3]). taken([wed,11,r3]). taken([wed,12,r3]). taken([wed,1,r3]).
taken([thurs,12,r4]). taken([thurs,1,r4]).
taken([fri,3,r1]). taken([fri,4,r1]).
```

You have to include all these atomic statements in your program. Only class-rooms and date/time not taken by other courses can be used to compute the timetable. The courses have somewhat different modes, where each "mode" consists of the

number of consecutive hours and the split between different dates of the week, if necessarily. For example, the mode "2+2" means that a class has to meet twice for 2 hours each time. The mode "2+1" means that the class has to meet twice a week, first for 2 hours and then, on a different date, for 1 hour. The mode "1+1+1" means that the class has to meet 3 times a week for 1 hour only each day, on 3 different dates. The task is to schedule classes for 6 courses: artificial intelligence (2+2), operating systems (2+1), web design (3+0), data structures (2+1), algorithms (2+1) and networks (1+1). There are a few constraints that have to be respected when we compute a timetable.

1. Since the professor who teaches operating systems is also an instructor for the course on networks, make sure that these classes do not meet at the same time. The predicate *different_time(List)* means that all time slots mentioned in $List$ are different. This is a stronger constraint than using the predicate *all_diff(List)* from the lecture notes. For example, the list *[tue,11,r1]* is different from the list *[tue,11,r4]*, but in terms of time slots mentioned here, they are not different. Write a program that implements this predicate and use it to formulate the constraint that time slots for operating systems and networks are different.

2. Similarly, one of the professors teaches both algorithms and data structures and for this reason, these courses cannot be taught at the same time, even if they are in different rooms.

To implement the constraint that the operating systems class meets on one day for 2 hours and on another day for 1 hour only, you can use the predicate *two_consecutive_hours([T1,T2])* similar to the predicate in notes. Also, implement the predicate *same_day(List)* meaning that all class times in *List* are on the same day. Use it to represent the fact that it is not true that all three class hours of the operating system class are on the same day (e.g., if the operating system class meets for 2 consecutive hours on Monday, then this class should not meet for 1 hour more on Monday, but it can meet for 1 hour on any other day). In addition, implement a helping predicate *available(List)* to express the fact that all class times in *List* are not taken by other courses. Use this predicate in your program. Finally, implement also the predicate *in_conflict(List1,List2)* meaning that class times from *List1* are in conflict with class times from *List2*. Two lists of classes are in conflict if they have a class in common, i.e., if there exists a class in the first list that is also a member of the second list. This conflict can happen only if two different courses are scheduled in the same room on the same day at the same time. We should avoid conflicts in any timetable. For example, use this predicate to say that the list of AI class times is not in conflict with any class time when the web design course is scheduled.

Solve this problem using smart interleaving of generate and test, as discussed in class. Keep this program at the top of the file `timetable.pl` Your program can compute several different solutions. The TA who will mark your assignment can provide an additional file with atomic statements to make sure that there is an unique solution. For this reason, be careful when you type (or copy) constants from atomic statements given to you. Do **not** include any additional atomic statements yourself. Write comments in your program file: **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, if you decide to use one. Find also how much time your program with interleaving takes to compute an answer: use `cputime` library predciate, but do not include time that your program spends on output/printing; only computation time should be included.

Write your session with Prolog to the file **timetable.txt**, showing the queries you submitted and the answers returned, including computation time. Make sure that your output is easy to read: print your solution using the predicate `write(X)` and `nl`, when needed. You must implement the `print_solution(List)` predicate similar to what we considered in class (see posted PDF slides for examples). Namely, it calls `solve(List)` and then formats output to make it easily readable. For example, here is one of the solutions that my program finds and prints.

```
?- print_solution(List).
List = [[[mon,9,r1], [mon,10,r1], [tues,9,r1], [tues,10,r1]],
[[mon,9,r2], [mon,10,r2], [tues,9,r3]], [[mon,11,r1], [tues,10,r3]],
[[mon,9,r3], [mon,10,r3], [tues,9,r4]], [[mon,11,r2], [mon,12,r2], [tues,11,r1]],
[[tues,12,r3], [tues,1,r3], [tues,2,r3]]]
AI course is scheduled on [mon, 9, r1]  [mon, 10, r1]  [tues, 9, r1]  [tues, 10, r1]
Operating Systems course is scheduled on [mon, 9, r2]  [mon, 10, r2]  [tues, 9, r3]
Networks course is scheduled on [mon, 11, r1]  [tues, 10, r3]
Data structures course is scheduled on [mon, 9, r3]  [mon, 10, r3]  [tues, 9, r4]
Algorithms course is scheduled on [mon, 11, r2]  [mon, 12, r2]  [tues, 11, r1]
Web design course is scheduled on [tues, 12, r3]  [tues, 1, r3]  [tues, 2, r3]
```

Handing in solutions: An electronic copy of your files **timetable.pl** and **timetable.txt** must be included in your **zip** archive.

3 (30 points). Peter, Romeo, Sam and Tom live on *distinct* floors of a high-rise building with 20 floors. One of them is an accountant, somebody is a businessman, somebody else is a computer scientist, and finally, someone is a dentist. Suppose the following facts are true:

1. Peter lives higher than Sam, but lower than Tom.
2. Romeo lives lower than the dentist.
3. Tom's apartment is 5 times higher than computer scientist's apartment.
4. Peter lives 3 floors lower than the dentist.
5. Computer scientist's apartment is not on the first 3 floors.
6. Were the businessman lived two floors lower, his apartment would be exactly in the middle between floors where the dentist and the accountant live.
7. Peter's apartment is on the same floor as businessman's apartment.
8. Sam's apartment is twice as lower as the dentist's apartment.
9. Were Romeo lived twice as higher than now, his apartment would be two floors below the apartment of the accountant.
10. The dentist lives either higher than Tom, or on the same floor with Tom.

Write a program that finds occupations and floors of all people mentioned in this story. Do not attempt to solve this puzzle yourself; your program must solve it. You have to submit your program in the file **logic.pl** More specifically, do the following.

- First solve this logical puzzle using smart interleaving of generate and test. You have to write a single rule that implements the predicate **solve1(L)**, and you can use helping predicates similar to those that we discussed in class. Of course, write also atomic statements that define a finite domain for all variables that you will introduce. As in the first question, you will need to be careful in your program regarding the order of constraints. Explain briefly the ordering you have chosen (write comments in you program). Determine how much computer time your computation takes using `cputime` library predicate.

  Finally, **output legibly** a solution that your program finds, so that when the TA who marks your assignment will run the query, he will be able to read easily if the solution is correct. Output your solution using the predicates `write(X)` and `nl`. You must print solution using a rule that implements the `print_solution(List)` predicate that we considered in class. Make sure this `print_solution(List)` predicate is implemented in your program, because the TA will use it as a query to test your program.

- Next, write a program that solves this problem using pure generate and test, without any smart interleaving: you have to implement the predicate **solve2(L)**. Determine how much time your computation takes using the following query:

  ```
  ?- Start is cputime, solve2(List),  End is cputime, Time is End - Start.
  ```

  (Warning: it can take several minutes depending on your computer. It takes about 6 minutes on my computer.) Keep this second program somewhere at the bottom of your file **logic.pl** There is no need to display a solution from this 2nd program using a formatted output.

- For both programs, copy your session with Prolog to the file **logic.txt**, showing the queries you submitted and the answers returned (including computation time). Write a brief discussion of your results in this file. Explain why the first program works much faster than the second one.

Handing in solutions: (a) An electronic copy of your file **logic.pl** must be included in your **zip** archive; (b) your session with Prolog, showing the query `solve1(List)` and the query `solve2(List)` that you submitted and the answers returned (the name of the file must be **logic.txt**); this file should also include computation time. Make sure that the solution your program outputs is easy to read.

How to submit this assignment. Read regularly *Frequently Answered Questions* and replies to them that are linked from the Assignments Web page at

`http://www.scs.ryerson.ca/~mes/courses/cps721/assignments.html`

If you write your code on a Windows machine, make sure you save your files as plain text that one can easily read on Linux machines. Before you submit your Prolog code electronically make sure that your files do not contain any extra binary symbols: it should be possible to load `puzzle2.pl` or `timetable.pl` or `logic.pl` into a recent release 6 of ECLiPSe Prolog, compile your program and ask testing queries. TA will mark your assignment using ECLiPSe Prolog. If you run any other version of Prolog on your home computer, it is your responsibility to make sure that your program will run on ECLiPSe Prolog (release 6 or any more recent release), as required. For example, you can run a command-line version of *eclipse* on moon remotely from your home computer to test your program (read handout about running *ECLiPSe Prolog*). To submit files electronically do the following. First, create a **zip** archive:

```
zip yourLoginName.zip puzzle1.pl puzzle2.pl timetable.pl timetable.txt logic.pl
```

where `yourLoginName` is the Ryerson login name of the person who submits this assignment from a group. Remember to mention at the beginning of each file *student, section numbers* and *names* of all people who participated in discussions (see the course management form). You may be penalized for not doing so. Second, upload your zip file

yourLoginName.zip

to D2L into "Assignment 3" folder. Make sure it includes **all** your files. Improperly submitted assignments will **not** be marked. In particular, you are **not** allowed to submit your assignment by email to a TA or to the instructor.

Revisions: If you would like to upload a revised copy of your assignment, then simply upload it again. (The same person must upload.) A new copy of your assignment will override the old copy. You can upload new versions as many times as you like and you do not need to inform anyone about this. Do not ask your team members to upload your assignment, because TA will be confused which version to mark: only one person from a group should submit different revisions of the assignment. The groups that submit more than one copy of their solutions will be penalized. The time stamp of the last file you upload will determine whether you have submitted your assignment on time.