# Cmpe 260 Prolog Project

Due Date: March 6, 2017 23:59 PM

## 1 Introduction

In this project, you are going to implement a scheduler for the final exams. As you all can guess, it is hard to manually assign times & locations for the final exams with the limited amount of time and classrooms; especially when there are so many students with so many conflicts. With your help, we may never have to deal with those conflicts again.

## 2 Knowledge Base

You will be given a knowledge base, which you will read (consult) from a file. Different knowledge bases can be consulted to test your approach.

### 2.1 student( StudentID, [ CourseIDs ] ).

Each student takes **at least 1 class** and can take **up to 7 classes**. Student count is kept very small here to make examples easier to understand. There will be many students in the actual knowledge base.

```
student(1415926, ['ec102', 'cmpe160', 'math102', 'math202']).
student(5358979, ['ec102', 'cmpe240', 'cmpe230', 'math202', 'phys201',
                  'ee212', 'math102']).
student(3238462, ['phys201', 'ec102', 'math102', 'math202']).
student(6433832, ['ec102', 'cmpe160', 'phys201']).
student(7950288, ['math102']).
```

### 2.2 available_slots( [ Slots ] ).

There are 3 slots in each day and 3 days have been given to keep examples easy, normally there are exams 6 days a week.

```
available_slots(['m-1', 'm-2', 'm-3', 'w-1', 'w-2', 'w-3', 'f-1', 'f-2', 'f-3']).
```

### 2.3 room_capacity( RoomID, RoomCapacity ).

The KB contains the available rooms and their capacities. Again, the actual KB will have more rooms. You can gather the list of the available rooms using room_capacity clauses. Room count and capacities given here are very small to keep examples clear.

```
room_capacity('nh101', 1).
room_capacity('nh105', 3).
room_capacity('ef106', 5).
```

## 3 Queries

For the project, you have to implement the following predicates.

### 3.1 clear_knowledge_base.

This predicate should clear all assertions from a KB (so that we can load a new one), and display some message to summarize what has been deleted.

### 3.2 all_students( −StudentList ).

This predicate should produce the list of all students in a given knowledge base.

Example for KB above:

```
?- all_students(StudentList).
StudentList = [1415926, 5358979, 3238462, 6433832, 7950288].
```

### 3.3 all_courses( −CourseList ).

This predicate should list all unique courses in given knowledge base.

Example for KB above:

```
?- all_courses(CourseList).
CourseList = ['ec102', 'cmpe160', 'math102', 'math202', 'cmpe240', 'cmpe230', 'phys201',
↪  'ee212'].
```

### 3.4 student_count( +CourseID, −StudentCount ).

This predicate should give student count of given course with `CourseID`.

Example for KB above:

```
?- student_count('math102', StudentCount).
StudentCount = 4.
```

### 3.5 common_students( +CourseID1, +CourseID2, −StudentCount ).

This predicate should give the count of students who take both of the courses given with `CourseID1` and `CourseID2`.

Example for KB above:

```
?- common_courses('cmpe160', 'math102', StudentCount).
StudentCount = 1.
```

### 3.6 final_plan( −FinalPlan ).

This predicate should calculate final exam time and locations without any conflict and report them as a list of lists that are in the following format: `[CourseID, RoomID, Slot]`. Predicate should continue to generate valid plans when ';' is pressed until all possible plans are found.

Example for KB above:

```
?- final_plan(FinalPlan).
FinalPlan = [['ec102', 'ef106', 'm-1'], ['cmpe160', 'nh105', 'w-2'], ['math102', 'ef106',
↪  'f-2'], ['math202', 'nh105', 'w-1'], ['cmpe240', 'nh101', 'm-3'], ['cmpe230',
↪  'nh101', 'f-3'], ['phys201', 'nh105', 'f-1'], ['ee212', 'nh101', 'w-2']].
```

### 3.7 errors_for_plan( +FinalPlan, −ErrorCount ).

This predicate should find errors in given `FinalPlan` and report their `ErrorCount`. When we test this predicate, we will not give any course name that do not exist in the KB. But given plan may not contain all courses in KB. So basically you **do not** have to check for "New" or "Missing" courses but you only should look for errors in given plan.

What is *Error*:

- For any two exams in same slot, error goes up by one for every student who takes both of them.
- For any exam that has been assigned to a classroom with smaller capacity than total number of attendee of corresponding course, error goes up by the amount: `NumberOfAttendee - RoomCapacity`
- You are not calculating error by student so intersection of different errors is not important to you. Ex: Assume that there is a conflict between 'ec102' and 'cmpe160' and there also is a student taking them both, and at the same time 'ec102' has 4 attendees in total, but has been assigned to 'nh105'. In this setup you should increase error count **by one for each error**, i.e. by 2 and not just 1. You cannot say "then student can go to exam 'cmpe160' and there will only be one error".

Examples for KB above:

```
?- conflict_for_plan([['ec102', 'ef106', 'm-1'], ['cmpe160', 'nh105', 'w-2'], ['math102',
↪  'ef106', 'f-2'], ['math202', 'nh105', 'w-1'], ['cmpe240', 'nh101', 'm-3'], ['cmpe230',
↪  'nh101', 'f-3'], ['phys201', 'nh105', 'f-1'], ['ee212', 'nh101', 'w-2']], ErrorCount).
ErrorCount = 0.
```

```
%% See 'cmpe160' and 'math102': there is a student who takes them both
?- conflict_for_plan([['ec102', 'ef106', 'm-1'], ['cmpe160', 'nh105', 'w-2'], ['math102',
↪  'ef106', 'w-2'], ['math202', 'nh105', 'w-1'], ['cmpe240', 'nh101', 'm-3'], ['cmpe230',
↪  'nh101', 'f-3'], ['phys201', 'nh105', 'f-1'], ['ee212', 'nh101', 'f-2']], ErrorCount).
ErrorCount = 1.

%% Previous conflict + 'ec102' has assigned to a small room. ErrorCount += (4 - 1)
?- conflict_for_plan([['ec102', 'nh101', 'm-1'], ['cmpe160', 'nh105', 'w-2'], ['math102',
↪  'ef106', 'w-2'], ['math202', 'nh105', 'w-1'], ['cmpe240', 'nh101', 'm-3'], ['cmpe230',
↪  'nh101', 'f-3'], ['phys201', 'nh105', 'f-1'], ['ee212', 'nh101', 'f-2']], ErrorCount).
ErrorCount = 4.
```

# 4 Documentation and Clarity

As documenting the code is essential for developing in the long term and as Prolog programs can be written in clever ways that cannot be easy to grasp at a first glance, you should document every predicate in your project and you will be graded for documentation of your code. A good code contains roughly the same amount of explanation in comments as code lines.

Besides documentation, it is also important to write code that is readable, so avoid small, clever, hard-to-comprehend hacks and unnecessary complexity in your code. You are also graded for code clarity. When you cannot find a clear way to handle something, explain how your code works in documentation, as it is important for us to grade your project.

# 5 Submission

You have to submit a single .pl file through Moodle with the file name STUDENTID.pl where you replace STUDENTID with your student id.

# 6 Some Tips on The Project

- Try to formalize the problem, specially the predicates that need to find a set, then try to convert the logic formula to Prolog.
- You can use `findall/3`, `bagof/3` or `setof/3`. Be careful when using `bagof/3` and `setof/3`, and remember to set which free variables to ignore.
- You can take a look at book, slides and 99 Prolog Problems.
- Try to build complex predicates over the simpler ones, the project is designed to encourage that.
- If a predicate becomes too complex; either divide it into some auxiliary predicates and implement them first, or take another approach.
- Use debugging, approach your program systematically.