CSCI1410 Fall 2018
Assignment 3: Knowledge Representation and Reasoning

Code Due Monday, October 1
Writeup due Thursday, October 4

1 Introduction

Someone has been taking George's Diet Dr. Pepper from his office and he needs to build an AI to help him find out who the thief is. He has decided to go back to college undercover and he's trying to decide which courses to take. However, George has mixed up all the courses and their prerequisites, so George doesn't know what to take. Help George figure out his courses so that he can find the thief!

In this assignment, we will represent the rules of courses and prerequisites in a knowledge base, implemented in Prolog.

2 Prolog

Prolog is a programming language that has its roots in first order logic. In this assignment, we will be using SWI-Prolog. **Below, you'll find all the information you need to complete the assignment**. Feel free to come back to this section after you've finished reading section 3!

2.1 Loading a Knowledge Base

To use Prolog, open up the terminal in a department machine and navigate to the directory where your knowledge base `curriculum.pl` is located. Type:

```
$ swipl
```

This should open up a prompt. Now, you can load your knowledge base as follows:

```
?- [curriculum].
```

Or, if you're using Windows, type:

```
?- make.
```

You should re-compile the knowledge base every time you make a change.

In order to exit SWI-Prolog, you can do the following:

```
?- halt.
```

2.2 Variables

In Prolog, variables start with an uppercase letter and constants with a lowercase letter. For example, in `singing(mark)`, `mark` is a constant while in `singing(X)`, `X` is a variable. We can use `is` to assign a value to a variable. For example,

```
isTwo(X):-
     X is 2.
```

2.3 Facts and Rules Syntax

Every fact, rule and query ends with a dot. We have provided you with a sample knowledge base, `example.pl`. You can play around with it and pass it some queries. The sample knowledge base is as follows:

```
singing(elon).
playingGuitar(mark).
playingGuitar(satya).
singing(jeff).

playingGuitar(elon):-
     happy(elon).

happy(mark):-
     singing(mark),
     playingGuitar(mark).

happy(satya):-
     singing(satya);
     playingGuitar(satya).
```

```
sad(satya):-
     not(playingGuitar(satya)).

grumpy(jeff):-
     singing(jeff)->
          false
     ;
          true.
```

2.3.1 If

In the above KB, the first four lines are facts and the rest are rules. Consider the rule on the fifth line:

```
playingGuitar(elon):- happy(elon).
```

The above line can be read as *elon* plays the guitar **if** *elon* is happy. The `:-` operator is used to denote "if" or "is implied by".

2.3.2 And

Consider the next rule:

```
happy(mark):-
     singing(mark),
     playingGuitar(mark).
```

The above can be read as *mark* is happy if *mark* is singing **and** playing the guitar. Thus, the `,` in Prolog is used to denote "and".

2.3.3 Or

Consider the next rule:

```
happy(satya):-
     singing(satya);
     playingGuitar(satya).
```

The above can be read as *satya* is happy if *satya* is singing **or** playing the guitar. Thus, the `;` in Prolog is used to denote "or".

2.3.4 Not

Consider the next rule:

```
sad(satya):-
    not(playingGuitar(satya)).
```

The above can be read as *satya* is sad if *satya* is **not** playing the Guitar. Thus, `not(...)` is used to denote negation.

2.3.5 If-then-else

Consider the last rule:

```
grumpy(jeff):-
     singing(jeff)->
          false
     ;
          true.
```

This can be read as, *jeff* is grumpy unless he's singing. The expression before the `->` is the expression that is checked for truth; if it's true, the next line is executed. If it's false, the line after the `;` is executed.

2.4 Operators

Since Prolog is built on true and false statements, you might find Boolean expressions useful! Prolog supports comparison operators, as illustrated in the table below:

Syntax	Explanation
`<Term1> @> <Term2>`	True if `Term1` is after `Term2` in the standard order of terms.
`<Term1> @<  <Term2>`	True if `Term1` is before `Term2` in the standard order of terms.
`<Term1> @=>  <Term2>`	True if `Term1` is after `Term2` in the standard order of terms, or both terms are equal.
`<Term1> @=<  <Term2>`	True if `Term1` is before `Term2` in the standard order of terms, or both terms are equal.
`<Term1> == <Term2>`	True if both terms are equal.
`<Term1> \== <Term2>`	True if terms are not equal.

2.5 Commenting

To add comments in Prolog, you can use `/* */` or %.

2.6 Querying the Knowledge Base

Once you've loaded the knowledge base, you can query it. Assume that the knowledge base contains two facts:

```
singing(elon).
singing(jeff).
```

Now, we can ask the knowledge base if a fact is true or false, as follows:

```
?- singing(elon).
?- true.
```

In this example, we've asked the knowledge base if a fact we've given it is true. We can also query for information that the knowledge base contains, as follows:

```
?- singing(X).
```

Prolog then returns:
`?- X = elon`

You'll notice that it's only returned one answer, while we know that there are more! To continue, simply type `;` after the first answer rather than pressing `enter`. Prolog will then return:

```
?- X = jeff
```

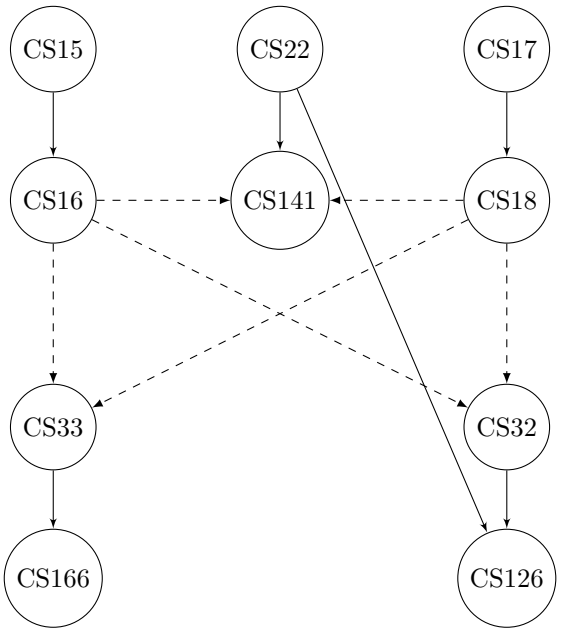
If you have any further questions about syntax, you can post on Piazza or come to TA hours!

Note: Every fact, rule and query in SWI-Prolog must end with a dot. If you forget the dot, you will run into errors!

3 Your Task

3.1 Knowledge Base

A knowledge base contains *facts* and *rules*. Your first task is to create a knowledge base describing courses and their prerequisites. The following is a graph of courses and prerequisites in the Computer Science department. Note that the graph does **not** strictly follow the Brown CS curriculum, meaning the rules of the graph are not necessarily the same as the Brown CS department's rules.



In the graph, a solid arrow from A to B means that A is a prerequisite of B. Dotted arrows mean that ONE of these courses is a prerequisite of the course. For example, CS15 is a prerequisite of CS16. However, to take CS141, you have to take CS22 and *either* CS16 or CS18. Similarly, to take CS32, you have to take *either* CS16 or CS18.

You also have the following rules about courses:

1. CS15, CS17, CS33, CS141 and CS126 are only offered in the fall.
2. CS16, CS18, CS22, CS32, CS166 are only offered in the spring.
3. A course is intro if it's offered in the fall and has no prerequisites, or if it's offered in the spring and has a prerequisite that has no prerequisites.
4. A course is intermediate if it is not an intro, but its prereqs are all intro classes.
5. A course is upper level if its prereqs are an intro **and** CS22, or if its prereqs are not intros.
6. Courses may only be of one level; i.e. intro courses may not also be intermediate courses, etc.
7. Course(s) can be neither a lower level nor an upper level course.

Please do not hard code these rules for each course. We will be adding new courses while testing and your knowledge base rules should work on them. Adding facts don't count as hard coding because facts need to reference specific information to be a fact rather than a rule. Hardcoding would be something like: intro(Course) :- Course == cs01;; Course == cs00 . (where the requirements to be intros haven't been taken into account at all).

While constructing your knowledge base, you **must** use the following predicates. We have specified which predicates you must write rules for, and which you must use as facts.

`fall(Course)`
Use: Fact
English version: True when `Course` is offered in the fall.

`spring(Course)`
Use: Fact
English version: True when `Course` is offered in the spring.

`has_prereqs(Course, Prerequisite)`
Use: Fact
English version: True when `Prerequisite` must be taken to be eligible to take `Course`, **or** if `Prerequisite` is one of multiple courses that can be taken to gain eligibility to take `Course`.

`no_prereqs(Course)`
Use: Fact
English version: True when `Course` can be taken without taking any other courses previously.

`intro(Course)`
Use: Rule
English version: True when `Course` is offered in the fall and has no prerequisites, or if it's offered in the spring and its prerequisite has no prerequisites.

`intermediate(Course)`
Use: Rule
English version: True when `Course` is not an intro, but its prereqs are intros.

`upper_level(Course)`
Use: Rule
English version: True if the prereqs of `Course` are not intros, or if its prereqs are CS22 and intros.

The courses are represented as `cs15`, `cs16`, `cs17`, `cs18`, `cs22`, `cs32`, `cs33`, `cs141`, `cs166` and `cs126`. You can use any additional predicates, but you must use the

ones specified above. We will be using these for testing purposes. You are given a stencil file `curriculum.pl` to fill in with the above rules and predicates.

3.2 Queries

Now that you've implemented the basic knowledge base, Mark has a specific set of questions that he wants the knowledge base to answer. In plain words, we have given you the queries that we will be testing you on. Your task is to add the appropriate facts to `curriculum.pl` so that your knowledge base answers each query correctly. **Although we won't be testing you on this explicitly, you should also figure out how to express each query in Prolog so that you can test your implementations.**

The first thing you should do is add one more rule to your knowledge base:

`can_take(Student, Course)`
Use: Rule
English version: True when `Student` is eligible to take `Course` (i.e. they have taken the necessary prerequisites.)

You must implement the `can_take` function to check a student's eligibility to take classes. We will be running our autograder using your version of `can_take`, so make sure it's correct! You will use the following fact to write your `can_take` rule:

`has_taken(Student, Course)`
Use: Fact
English version: True when `Student` has taken `Course`.

You now should now translate the following statements into Prolog by adding facts to your knowledge base using the `has_taken` predicate. Please use the indicated name in `this font` for each of the scenarios, or else our autograder won't work!

1. `mark` has only taken one intermediate class (and its prerequisites) and cannot take any more classes than he could without it.
2. `elon` can take `cs32` and `cs18`. You should assume he cannot repeat classes.
3. `sheryl` Sandberg is eligible to take all upper level classes.
4. The only class `jeff` Bezos can take that he hasn't already taken is CS32.

3.3 Prolog Output Clarifications

1. When making a query, it is okay if courses show up multiple times.

2. If a query returns true (as desired), but you continue it with a semicolon, it may return false. This is because Prolog returns false when the query cannot search for more possible solutions. As long as there is one true, this is okay.

4 Written Questions

Answer these in a numbered, typed document.

1. Given that the following statements are true, derive the truth value of A.

   $\neg D \wedge E$
   $C \vee D$
   $B \implies \neg C$
   $A \vee B$

2. "If it rains and you don't open your umbrella, you will get wet". Translate this statement to propositional logic and write its truth table.

3. Write the following statements in first order logic. You are given the functions: `AuthorOf(A, B)`, `IsBook(A)`, and `Equals(A, B)`.

   (a) All books have an author.

   (b) Sheryl Sandberg wrote the book *Lean In*.

   (c) At least one book has exactly three authors.

   (d) No books were written by Mark.

4. Discuss the differences between Propositional Logic and First Order Logic.

5 Grading

For part 3.1, we will be testing your KB by adding our own courses and ensuring that your rules return what we expect. This is why you cannot hardcode the rules we tell you to implement - i.e. you cannot say the Prolog version of, "A course is an intro if it is courseA or courseB or courseC" because when we add courseD to your KB, it should also tell us it's an intro.

In other words, you only need to make your KB work for the scenarios we have illustrated in the graph. For example, to test that you haven't hardcoded things, we might create a new intro sequence that is functionally the same thing as an intro pair that's in the graph, and ensure that your KB tells us those new courses are intros. But we aren't going to test some edge case that isn't logically represented by the graph and rules we gave you.

For part 3.2, the purpose is to think about what the Prolog query will be translated from English, and then add the appropriate facts to your KB. Note that

this means we are not formally testing your can_take rule. We expect your can_take rule to be functional enough so that our queries work with it. You should not interpret this as a license to hard code the answers to the queries in your can_take function, as it probably won't work. But also, if you feel that there's no other way to represent a scenario by thinking extensibly and sticking to what's in the handout, then you may want to consider what the special cases are and adjust your can_take function as necessary.

You can check `rubric.txt` in your stencil folder for more details about grading.

6 Handin Instructions

You must turn in `curriculum.pl` via the handin script. This is the only file we will be testing, so answer all of the Prolog questions in this file.

As usual, you must submit your answers to the comprehension questions via Gradescope.