CS 335 - Homework 5

Deadline:

Due by 11:59 pm on Friday, March 11

How to submit:

Submit your files for this homework using `~st10/335submit` on nrs-labs, with a homework number of `5`

Purpose:

To start to get comfortable with logic programming in Prolog

Important notes:

- Download an appropriate version of SWI-Prolog (http://www.swi-prolog.org) and install it, or use the version installed in BSS 313. (It *should* be installed in BSS 317, LIBR 121, and LIBR 122 as well.) Then, start it up.
  - If you are using this program in Windows, here is a very useful tip from the Spring 2008 section of CS 335: start SWI-Prolog, run:

    ```
    protocol('some_bizarre_name.txt').
    halt.
    ```

    ...and then search to find out the default directory for that SWI-Prolog installation.

    If you now place your file of Prolog facts and rules such as `my_file.pl` there, then

    ```
    ['my_file.pl'].
    ```

    ...should successfully load those facts and rules into your SWI-Prolog session.
  - (Although Mr. Frasche (Spring 2008) also noted that, at least from the command line, starting up SWI-Prolog with either:

    ```
    swipl -s my_file.pl
    swipl -f my_file.pl
    ```

    ...does start it up and load in that file's facts and rules in one fell swoop, and I can confirm that these work on my Mac OS X version as well. The `swipl` manual page under Unix says that the `-s` is used to load the given file as a script, whereas `-f` is used to use the given file as the initialization file instead of `.plrc`. I'm not sure of the practical difference between these since currently we aren't using a `.plrc` file, but I suspect in the long term using `-s` is the better/more robust habit...)
- Remember that if you type the predicate `protocol` followed by a constant that is surrounded by

single quotes, then everything that you do and see from that point is saved in a file of that name. That is,

```
?- protocol('ex.txt').
true.

?- halt.
```

...results in a file `ex.txt` in the current directory that contains

```
true

?- halt.
```

(although the Windows version appears to vary slightly, for example maybe saying `Yes` instead of `true`, and maybe with slightly different formatting/blank lines). (So, `protocol/1` is true if a file with the given name can be opened for writing, and in this case it also has this desired side-effect.)

- You will use `protocol` to generate example Prolog program runs and queries.

- If a query may have multiple ways it can be proven true (via more that one variable unification), then if you type `;` (semicolon) after the first result, Prolog will try to find another way to prove it true. You can keep typing `;` as long as you want to keep seeing additional ways that query may be proven to be true. When (or if) you don't wish to see any more, type a `.` (period) instead to stop.
  - (Remember: in Prolog, `,` (comma) can be thought of as logical AND, and `;` can be thought of as logical OR.)
- Recall that the arity -- the number of expected arguments -- of a predicate is often indicated by following the predicate name with a slash and that number. So, for example, one might write `halt/0` (since `halt` has no arguments) or `protocol/1` (since `protocol` expects one argument).
  - Here is an additional practical benefit to knowing this notation: if you would like to see the listing for a loaded rule from within your `swipl` session, you can, by using the predicate `listing/1` whose expected argument is the predicate written using this notation.
  - That is, if you had loaded the facts and rules from our Week 7 Lecture 1 little tiny Prolog knowledge base `weather.pl`, then you could get a listing of the predicate `snowy`, that expects one argument, by typing the query:

    ```
    ?- listing(snowy/1).
    ```

    You would then see something like this:

    ```
    snowy(A) :-
            rainy(A),
            cold(A).

    true.
    ```

The Problems:

Problem 1

Start a Prolog knowledge base `hw5-1.pl`. The first thing in this file should be comments containing at least:

- your name
- `CS 335 - Homework 5 - Problem 1`
- and the last-modified date

This problem will simply allow you to demonstrate that you can load a Prolog knowledge base and execute Prolog queries on that knowledge base.

1 part a

Consider that Week 7 Lecture 1 Prolog knowledge `weather.pl`. Within `hw5-1.pl`:

- copy in the contents of `weather.pl`
- add at least 10 more analogous weather-related facts of your choice, including at least two predicates different from those already there
- add at least 2 more weather-related rules that can be used reasonably with your resulting set of weather-related facts
- NOTE: remember that `swipl` is MUCH happier if you group instances of the SAME predicate together (e.g., notice how the two `rainy` facts are next to each other in `weather.pl`.)

1 part b

Now start up `swipl`, and:

- use `protocol` to start saving your Prolog session into a file `hw5-1-ex.txt`
- now load your Prolog knowledge base `hw5-1.pl`
- use the `listing` predicate appropriately to display a listing of each of the rules in this Prolog knowledge base.
- for at least two of the predicates used in facts in this Prolog knowledge base, write at least one query that:
  - can be proven true
  - includes at least one variable
  - (keep typing `;` to generate all of the variable assignments /unifications that can be used to prove your query true)
- for each of the rules in this Prolog knowledge base, write at least one query that:
  - can be proven true
  - includes at least one variable

  - (keep typing `;` to generate all of the variable assignments/unifications that can be used to prove your query true)
- write at least 3 queries of your choice that **cannot** be proven true using this Prolog knowledge base.

Submit your resulting `hw5-1.pl` and `hw5-1-ex.txt`

Problem 2

Start a Prolog knowledge base `hw5-2.pl`. The first thing in this file should be comments containing at least:

- your name
- `CS 335 - Homework 5 - Problem 2`
- and the last-modified date

Recall that, in Prolog, if you use a single _ (underscore) as a variable, it is considered to be "don't care", and can unify with anything. (In fact, unlike other variables, multiple _'s in the same rule can match different things!) But Prolog will not bother to tell you what a _ unified with. (As the J. R. Fisher tutorial notes in its Section 2-3, "Prolog allows these variables to freely match any structure, but no variable binding results from this gratuitous matching.")

Here is a different version of `has_ancestor/2` demonstrating don't care variables in action:

```
/* has_parents(Child, Mother, Father) is meant to mean that Child has
   parents Mother and Father
*/

has_parents(jane, sally, bob).
has_parents(john, sally, bob).
has_parents(sally, mary, al).
has_parents(bob, ann, mike).
has_parents(mary, lee, joe).

/* has_ancestor(Person, Ancestor) is meant to mean that Person has
   ancestor Ancestor
*/

has_ancestor(Person, Ancestor) :- has_parents(Person, Ancestor, _).
has_ancestor(Person, Ancestor) :- has_parents(Person, _, Ancestor).
has_ancestor(Person, Ancestor) :- has_parents(Person, Someone, _),
                                  has_ancestor(Someone, Ancestor).
has_ancestor(Person, Ancestor) :- has_parents(Person, _, Someone),
                                  has_ancestor(Someone, Ancestor).
```

This also is an example of recursion in Prolog -- the `has_ancestor/2` predicate is defined recursively via the four rules shown:

- Person has ancestor Ancestor if Person has parents Ancestor and anyone; or,
- Person has ancestor Ancestor if Person has parents anyone or Ancestor; or,

- Person has ancestor Ancestor if Person has parents Someone and anyone and Someone has ancestor Ancestor
- Person has ancestor Ancestor if Person has parents anyone and Someone and Someone has ancestor Ancestor

You should be able to see that the `has_ancestor/2` predicate happens to have two base cases and two recursive cases.

2 part a

Put these facts and rules into the knowledge base `hw5-2.pl`.

2 part b

Use `protocol` to create a record of a `swipl` session in a file `hw5-2-ex.txt` in which you:

- load your `hw5-2.pl` knowledge base
- write a query whose result could answer the question: who are bob's parents?
- write a query whose result could answer the question: who are john's ancestors?
- write a query whose result could answer the question: who has ann as an ancestor?
- write at least one more query whose results you find interesting.

Submit your resulting `hw5-2.pl` and `hw5-2-ex.txt`.

Problem 3

(adapted from C. Collberg's Arizona State CSc 372 Assignment #4, Fall 2005)

Start a Prolog knowledge base `hw5-3.pl`. The first thing in this file should be comments containing at least:

- your name
- `CS 335 - Homework 5 - Problem 3`
- and the last-modified date

3 part a

Consider this knowledge base of facts, describing what something is made up of. Add these to `hw5-3.pl`

```
/* has(Thing, Part, Quantity) is meant to mean that Thing has
   that Quantity of that Part
*/

has(bicycle, wheel, 2).
has(bicycle, handlebar, 1).
has(bicycle, brake, 2).
has(wheel, hub, 1).
has(wheel, spoke, 32).
```

```
has(bicycle, frame, 1).
has(brake, pad, 1).

has(car, steering_wheel, 1).
has(car, stereo, 1).
has(car, tire, 4).
has(car, brake, 4).
```

3 part b

Use `protocol` to create a record of a `swipl` session in a file `hw5-3b-ex.txt` in which you:

- load your `hw5-3.pl` knowledge base
- write a query that you could use to bind a variable of your choice to the number of spokes in a wheel
- write a query that you could use to bind two variables of your choice to each part and quantity of that part a car has (with successive typings of ;).

3 part c

To your knowledge base `hw5-3.pl`, add a rule `includes/2`, `includes(Piece, Thing)` that succeeds (can be proven) if `Piece` is included within `Thing`. That is, `includes/2` should behave like so:

```
?- includes(spoke, wheel).
true ;
fail.

?- includes(spoke, bicycle).
true ;
fail.

?- includes(spoke, car).
fail.
```

`includes/2` can also be used to enumerate the pieces that make up a thing, or of which a piece is part:

```
?- includes(Piece, bicycle).
Piece = wheel ;
Piece = handlebar ;
Piece = brake ;
Piece = frame ;
Piece = hub ;
Piece = spoke ;
Piece = pad ;
fail.

?- includes(spoke, Thing).
Thing = wheel ;
```

```
Thing = bicycle ;
fail.
```

Once you are happy with your rule `includes/2`, use `protocol` to create a record of your swipl session in a file `hw5-3c-ex.txt` in which you:

- load your revised `hw5-3.pl` database, now including `includes/2`
- run the above five queries to demonstrate that your `includes/2` behaves as it should
- add at least 2 additional queries of your choice that you think show something interesting.

Submit your resulting `hw5-3.pl`, `hw5-3b-ex.txt`, and `hw5-3c-ex.txt`.

Problem 4

Imagine a fictitious social networking site where one links to people they are fans of -- these links, then, are uni-directional; for example, I might be a fan of Tony Kornheiser, but he, not having the foggiest notion of who I am, is not a fan of me. Tony is a fan of Michael Wilbon, however.

Say that predicate `fan_of/2` indicates such a relationship; the following facts, then, could correspond to what I have described above:

```
fan_of(sharon_tuttle, tony_kornheiser).
fan_of(tony_kornheiser, michael_wilbon).
```

4 part a

Write at least 10 facts involving the predicate `fan_of/2`, making sure that:

- at least some of the facts include people who are fans of people who are also fans of someone else
- at least one person is a fan of himself/herself

Create a Prolog knowledge base named `hw5-4.pl` containing these facts.

4 part b

Write a rule `one_degree/2` which has two parameters, and which is true if someone is a direct fan of someone else (besides himself/herself); add this to `hw5-4.pl`.

4 part c

How about "two degrees" of separation? Make a rule `two_degrees/2` which has two parameters, and which is true for two people P1 and P2 if P1 is a fan of someone who is a fan of P2. (Note that the intent is that that "someone" is not the same person as P1 or P2, although it *might* be the case that P1 and P2 could be the same person... 8-)) Add this rule to `hw5-4.pl`.

4 part d

And how about "three degrees" of separation? Make a rule `three_degrees/2` which has two parameters, and which is true for two people P1 and P2 if P1 is a fan of "someone-1" who is a fan of "someone-2" who is a fan of P2 (Note that the intent here is that "someone-1" and "someone-2" are not the same person, that P1 is not the same as "someone-1", and "someone-2" is not the same as P2) Add

this rule to `hw5-4.pl`.

4 part e

Once you are happy with your rules and facts, use `protocol` to create a record of a `swipl` session in a file `hw5-4-ex.txt` in which you:

- load your `hw5-4.pl` knowledge base,
- run at least two queries involving just the `fan_of/2` predicate such that:
  - each includes at least one variable,
  - at least one such query has at least two different (sets of) variable bindings, and
  - at least one such query has no variable bindings.
- run at least two queries **each** involving the rules `one_degree/2`, `two_degrees/2`, and `three_degrees/2`, such that:
  - each involves at least one variable,
  - each has at least one variable binding, and
  - at least one for each of the rules has more than one variable binding.
  - (Note that this is a total of at least **six** queries.)
- add at least 2 additional queries of your choice that you think show something interesting.

Submit your final versions of `hw5-4.pl` and `hw5-4-ex.txt`.

Problem 5

Inspired by http://www.amzi.com/AdventureInProlog/ tutorial, section 1, "Getting Started"

Consider: some Prolog built-in predicates always (or almost always) succeed, but have a handy side-effect along the way. `protocol/1` is such a predicate that we have already used frequently.

If you were to want to write a Prolog application, you might want to print something to the screen. The `write/1` predicate writes the value of its argument to the screen and is true; `writeln/1` is similar, except it also writes a newline to the screen after the value of the argument. (You can get just a newline written to the screen using the predicate `nl/0`.)

```
tryit1 :- write('Hello'), write('World').

tryit2 :- writeln('Hello'), write('World').

?- tryit1.
HelloWorld
true.

?- tryit2
Hello
World
true.
```

We will be using `trace/0` to see what is happening in Prolog's resolution and backtracking; if you were wondering whether Prolog could have an equivalent to a "debugging cout", note that you could use `write/1` and `writeln/1` in this manner (note how this also demonstrates that it is okay to load in more than one knowledge base within a `swipl` session):

```
/* play-write.pl */

show_it(Person) :- female(Person),
                   writeln(Person),
                   has_parents(Person, _, _).
```

```
?- ['queen-v4.pl'].
% queen_v4.pl compiled 0.01 sec, 5,480 bytes
true.

?- ['play-write.pl'].
% play_write.pl compiled 0.00 sec, 824 bytes
true.

?- show_it(Who).
amelia
victoria_s_c
victoria
Who = victoria ;
alice
Who = alice.
```

So, what is happening with this query `show_it(Who).`?

- `female(amelia)` can be proven, so `writeln(Person)` writes `amelia`, but `has_parents(amelia, _, _)` cannot be proven, so Prolog backtracks.
- `female(victoria_s_c)` can be proven, so `writeln(Person)` writes `victoria_s_c`, but `has_parents(victoria_s_c, _, _)` cannot be proven, so Prolog backtracks.
- `female(victoria)` can be proven, so `writeln(Person)` writes `victoria`, and `has_parents(victoria, _, _)` can be proven, so `show_it(Who)` can be proven with `Person = victoria`
- if one now types semicolon (try to prove it again), now it tries `female(alice)`, which can be proven, and so `writeln(Person)` writes `alice`, and `has_parents(alice, _, _)` can be proven, so `show_it(Who)` can be proven with `Person = alice`
- and it is done.

J. R. Fisher's Prolog tutorial, Section 4.2, notes that the so-called "Universal" predicate `true/0` always succeeds as a goal, and "Universal" predicate `false/0` always fails as a goal.

With that information, consider the following rule, inspired by an example in the "Adventure in Prolog" tutorial:

```
mystery :-
   writeln('What is this?'),
```

```
    has_parents(_, X, Y),
    write(X), write(' and '), writeln(Y),
    fail.
```

The knowledge base `queen-v4.pl` from lecture is posted along with this homework assignment. Copy it into `hw5-5.pl` and add rule `mystery` to the end of it. Then...

5 part a

Start up a swipl session, and use `protocol` to create a record of your swipl session in a file `hw5-5-ex.txt` in which you:

- load your `hw5-5.pl` knowledge base,
- and try the query:

```
?- mystery.
```

5 part b

In a plain text file `hw5-5b.txt`, describe the query `mystery.`'s results. (That is, what is it resulting in? What is it doing? You *don't* have to explain *how* it is doing it -- but *what* is it doing?)

Problem 6

What if you would like to add and remove things from a Prolog knowledge base? You only need a few pieces to get started with this:

- Predicates that might have facts added and retracted must be indicated as **dynamic** at the beginning of a Prolog knowledge base. The "Adventure in Prolog" tutorial notes that:

```
:- dynamic F/A
```

  - "The `dynamic/1` directive specifies that the clauses of the predicate with functor F and arity A will be stored as dynamic. This directive is only needed by the compiler, to let it know which predicates NOT to compile. ...
  - Multiple comma separated F/A pairs can be entered with a single dynamic directive."

So, if you know that you might want to assert and retract instances of a certain predicate, you include this directive -- a Horn clause with a right-hand-side but not left-hand-side, interestingly enough.

Say that I might want to do just this with predicates `is_place/1` and `now_at/1`. Then I could use the directive:

```
:- dynamic is_place/1, now_at/1.
```

- And, you can assert and retract instances of such dynamic predicates using `assert/1` and `retract/1`:
  - `assert( is_place(blue_lake) ).`
  - `retract( now_at(arcata) ).`

With those pieces, you should have what you need to dynamically change a knowledge base during a `swipl` session. Consider the following example of these in a knowledge base -- it is also posted along

with this homework handout, as the file `hw5-prob6.pl`:

```
:- dynamic is_place/1, now_at/1.

is_place(arcata).
is_place(eureka).
is_place(mckinleyville).

now_at(arcata).

make_place(Place) :- \+( is_place(Place) ),
                     assert( is_place(Place) ).

go_to(Place) :- is_place(Place),
                retract( now_at(_) ),
                assert( now_at(Place) ),
                write('I am now at '),
                writeln(Place).

where_am_i :- write('I am at '),
              now_at(X),
              writeln(X).

list_places :- writeln('current places are: '),
               is_place(Place),
               writeln(Place),
               fail.
```

Create a copy of `hw5-prob6.pl` in your current working directory for your use. Then start up a `swipl` session, load this `hw5-prob6.pl` knowledge base, and use `protocol` to create a record of a `swipl` session in a file `hw5-6-ex.txt` in which you:

6 part a

Run a query that will allow one to determine all of the current `is_place/1` facts. (There is more than one way to do this -- choose any of these that works.)

6 part b

Run a query that will allow one to determine the current `now_at/1` fact. (There is more than one way to do this -- choose any of these that works.)

6 part c

Run a query (but **not** using `assert/1` or `retract/1`) that will, after it is tried, result in there being a different `now_at/1` fact in the knowledge base than there was before.

6 part d

Run a query (but **not** using `assert/1`) that will, after it is tried, result in there being an additional `is_place/1` fact that was not in the knowledge base before this.

6 part e

Run your query from part a again (although you may use one of the other ways of doing the same thing, if you would like.) If part d was successful, you should see your new `is_place/1` fact amongst the results of this query.

6 part f

Run your query from part b again (although you may use one of the other ways of doing the same thing, if you'd like.) If part c was successful, you should see your different `now_at/1` fact as the result of this query.

6 part g

Have you written queries involving all of the rules in this knowledge base yet? If not, write additional query/queries until you have.

6 reflection

It is your choice if you try this as part of the session that you submit for Problem 6. This part does **not** have to be turned in, but you are responsible for knowing the answers: what is the result of the query:

```
?- go_to(X).
```

...if you keep typing ; for as long as you can? What is the result of the query

```
?- where_am_i.
```

...after you have done so?

Problem 7

This was adapted from:
http://www.csc.vill.edu/~dmatusze/8310summer2001/assignments/adventure.html

For this problem, you will start writing an adventure game in SWI-Prolog. Pick any theme you like for your adventure game: rescue, survival, treasure hunt, "a day in the life," historical, science fiction, literature, or whatever else appeals to you.

Use the file `game0.pl` available along with this homework handout and use it as a starting point. This is an absolutely boring game consisting of one room, one object, and one direction you can go (but going in that direction takes you back to the same room). Add to this code to create your own game; if it doesn't do what you want, fix it so that it does. This is free code, to use or modify any way you like. If you don't want to use it, that's OK too.

Your program should eventually contain at least one (or more, if you like) of **EACH** of the following:

- Locked door
  - In its most boring form, you must find a key and use it to unlock a door, thus giving you access to one or more additional rooms.
  - With a little more imagination: You aren't admitted without a badge. You need to buy a ticket. You must give the troll a gold piece before you can cross the bridge. Waving the magic wand causes the rainbow bridge to appear. Et cetera. Any sort of locked door puzzle will do.

- Hidden object
  - Boring form: You open a box and find something inside.
  - More interesting: You break open a treasure chest. You use the combination to a safe. You peer into the crystal ball. You buy the candy bar from the vending machine. You disassemble the robot to get some part out of it.
- Incomplete object
  - Your flashlight needs batteries. Your gun needs bullets. Your car needs gas. Your bicycle has a flat tire. You need a computer to get at the information on a CD-ROM.
- Limited resources
  - You have a limited amount of time (to find the bomb before it goes off) or money (to buy the things you need), or food, drink, or sleep (so you don't collapse), or some other resource. Maybe you can find more resources in the game, maybe you can't. Depending on just what you decide to do, you may want to use arithmetic in Prolog. (We should discuss arithmetic in Prolog in class at some point, or you can read up on it in one of the posted Prolog tutorials.)

You should have a `start/0` predicate (similar to the one provided) that I can use to start your game and find out what commands you have added. **Don't make me look at the code to figure out how to run your program!**

You'll be working on this for more than one week; by **this** homework's deadline, you need to submit working attempts for at least **ONE** of the four requirements locked-door/hidden-object/incomplete-object/limited-resources.

Eventually, you will be expected to submit working attempts at (eventually) all FOUR of the requirements locked-door/hidden-object/incomplete-object/limited-resources as part of one or more future homeworks. (Some additional Prolog features will need to be used in future versions of your game, also.)

You need to submit (for Homework 5 Problem 7):

- your knowledge base thus-far (in a file named `hw5-game1.pl`),
- a transcript of a sample run of your program (in a file named `hw5-game1-sample.txt` created by the `protocol` predicate)
- a `hw5-game1-readme.txt` file that briefly describes your game, and in particular briefly describes your created-so-far locked-door/hidden-object/incomplete-object/limited-resource.

NOTE: you should plan to demonstrate 3-5 minutes worth of your favorite aspects of your game-so-far in class in some class session after the Homework 5 deadline, so that you can give each other ideas/inspiration as you proceed. This demonstration will be worth some quantity of clicker-question points to be announced.