CSC 372, Spring 2016
Assignment 8
Due: Friday, April 15 at 23:59:59

Game plan for the Prolog assignments

Our work with Prolog will be distributed across three assignments, with the following due dates:

Assignment 8	Friday, April 15
Assignment 9	Friday, April 22
Assignment 10	**Wednesday**, May 4

Remember that the video assignment is also due on May 4.

The Usual Stuff

Make an `a8` symlink that references `/cs/www/classes/cs372/spring16/a8`. Test using `a8/tester` (or `a8/t`).

Use SWI Prolog!

We'll be using SWI Prolog for the Prolog assignments. On lectura that's `swipl`.

About the `if-then-else` structure (`->`) and disjunction (`;`)

To encourage thinking in Prolog, you are strictly prohibited from using the `if-then-else` structure, which is represented with `->`. (Section 4.7 in Covington talks about it.)

Disjunction, represented with a semicolon (`;`), is occasionally very appropriate but it's easy to misuse and make a mess. Section 1.10 in Covington talks about it. Here's the rule for us: If you think you've found a good place to use disjunction, ask me about it; but unless I grant you a specific exemption, you are not allowed to use disjunction. (My general rule is this: don't use disjunction unless it avoids significant repetition.)

Easy Money!

Due to the time frame for this assignment and not wanting to underweight problems on assignments 9 and 10, I think you'll find that the time required to do this assignment is a bit low with respect to the points assigned.

Problem 1. (7 points, ½ point each) `queries.pl`

For this problem you are to write a number of queries, packaged up as rules. Some are easier than their half-point and some are harder, but they're all worth a half-point.

`a8/queries-starter.pl` starts like this:

```
% What foods are green?
q0(Food) :- thing(Food,green,yes).

% What are all the things?
```

```
q1(Thing) :- true.

% What are the colors of non-foods?
q2(Color) :- true.
```

q0 above is a completed example. The comment just prior specifies a question, "What foods are green?" Following that comment is a query that will answer that question. Let's load up the file and try `q0`:

```
$ swipl a8/queries-starter.pl
[...lots of singleton warnings due to the uncompleted queries...]
...
?- q0(F).
F = broccoli ;
F = lettuce.
```

Your task is to replace the dummy bodies (just `true`) for all the rules. The first few use the facts in `a8/things.pl`; the rest use the facts in `a8/fcl.pl`. Begin by copying `a8/queries-starter.pl` to `queries.pl`, and then edit `queries.pl`.

When your `queries.pl` is complete you should see behavior like this:

```
$ swipl queries.pl
...

?- q1(T).
T = apple ;
T = broccoli ;
...
T = stopsign ;
T = bagel.

?- q2(NF).
NF = brown ;
NF = green ;
NF = blue ;
NF = red.
```

Leave the sample rule q0 in place—the tester uses it.

The `:-QUERY.` construct

You'll see that `a8/queries-starter.pl` ends like this:

```
:-[a8/fcl].
:-[a8/things].
```

When consulting a file, Prolog assumes that it contains clauses that constitute the knowledgebase but sometimes we want to execute queries when consulting a file. The construct **`:-`*`QUERY`*`.`** indicates that *`QUERY`* is to be performed.

The two lines above cause `a8/fcl.pl` and `a8/things.pl` to be consulted, providing the facts to be used by this problem's queries.

Grading

<u>When grading I'll use altered versions of `a8/things.pl` and `a8/fcl.pl`, with facts added, deleted, and changed.</u> Write queries to be general, rather than "wired" for current data. For example, if something involves the cost of an orange, use a goal like `cost(orange, OC)` to get the cost of an orange rather than visually inspecting `a8/fcl.pl`, seeing `cost(orange,3)`, and using 3 for the cost of an orange.

<u>Important:</u> The usual guarantee of 75% of the points for passing all supplied test cases does not apply for this problem.

A note about the tester

You'll see that the tester uses a Prolog query with several goals for the rules in `queries.pl`:

```
findall(X,q1(X),L), sort(L,Results), writeln('Results:'),
member(X,Results), writeln(X), fail.
```

We'll be learning about `findall`, `sort`, and `member` soon, but briefly, here is what's happening: `findall` makes a list of all results produced by `q1(X)` and then `sort` sorts them, <u>removing duplicates</u>. The `member(X,Results), writeln(X), fail` sequence causes the results to be written out, one per line.

Problem 2. (1 point) `altrules.pl`

The first examples we saw with Prolog involved `food/1` and `color/2` facts. Then on slide 48 we saw an alternate representation of the same data using `thing/3`.

For this problem you are to implement `food/1` and `color/2` as rules that use the `thing/3` facts.

Your solution should look like this:

```
:-[a8/things].

food(F) :- ...
color(T,C) :- ...
```

The first line consults `a8/things.pl`.

Your task is to simply fill in the bodies for the `food` and `color` rules.

Usage:

```
$ swipl altrules.pl
...
?- food(X).
X = apple ;
X = broccoli ;
...

?- color(apple,red).
true.

?- color(F,green).
F = broccoli ;
```

```
F = grass ;
F = lettuce.

?-
```

Problem 3. (2 points) `sequence.pl`

Write a predicate `sequence/0` that outputs the sequence below.

```
?- sequence.
10101000
10101001
10101010
10101011
10111000
10111001
10111010
10111011
true.
```

Be sure that `sequence` produces `true` when done, as shown above.

Two notes: (1) Don't over think this one. (2) **Don't just "wire-in" the output verbatim**, like `writeln(10101000), writeln(10101001), ...`—**that'll be a zero!**

Problem 4. (7 points) `rect.pl`

In this problem you are to implement several simple predicates that work with `rect(width,height)` structures that represent position-less rectangles having only a width and height.

`square(+Rect)` asks whether a rectangle is a square.

```
?- square(rect(3,4)).
false.

?- square(rect(5,5)).
true.
```

`landscape(+Rect)` is true iff (if and only if) a rectangle is wider than it is high. `portrait` tests the opposite—whether a rectangle is higher than wide. A square is neither landscape nor portrait.

```
?- landscape(rect(16,9)).
true.

?- landscape(rect(3,4)).
false.

?- portrait(rect(3,4)).
true.

?- portrait(rect(10,1)).
false.

?- landscape(rect(3,3)).
false.
```

```
?- portrait(rect(3,3)).
false.
```

`classify(+Rect,-Which)` instantiates `Which` to `portrait`, `landscape` or `square`, depending on the width and height. If `Rect` is not a two-term `rect` structure, then `Which` is instantiated to `wat`.

```
?- classify(rect(3,4),T).
T = portrait.

?- classify(rect(10,1),T).
T = landscape.

?- classify(rect(3,3),T).
T = square.

?- classify(rect(3),T).
T = wat.

?- classify(10,T).
T = wat.
```

You may need to use some cuts (slide 109+) to prevent `classify` from producing bogus alternatives. **Here is an example of BUGGY behavior:**

```
?- classify(rect(5,7),T).
T = portrait ;    First answer is correct but there should be no alternatives!
T = square ;
T = wat.
```

Needless to say, use your `portrait/1`, `landscape/1`, and `square/1` predicates to write `classify/2`.

`rotate(?R1,?R2)` has three distinct behaviors:
(1) If `R1` is instantiated and `R2` is not, `rotate` instantiates `R2` to the rotation of `R1`.
(2) If `R2` is instantiated and `R1` is not, `rotate` instantiates `R1` to the rotation of `R2`.
(3) If both are instantiated, `rotate` succeeds iff `R1` is the rotation of `R2`.

Examples:

```
?- rotate(rect(3,4),R).
R = rect(4, 3).

?- rotate(R,rect(3,4)).
R = rect(4, 3).

?- rotate(rect(5,7),rect(7,5)).
true.

?- rotate(rect(3,3),R).
R = rect(3, 3).
```

`rotate` should also handle cases like these:

```
?- rotate(rect(3,4),rect(W,H)).
```

```
W = 4,
H = 3.

?- rotate(rect(3,X),rect(Y,4)).
false.
```

`smaller(+R1,+R2)` succeeds iff both the width and height of `R1` are respectively less than the width and height of `R2`. Rotations are not considered.

```
?- smaller(rect(3,5), rect(5,7)).
true.

?- smaller(rect(3,5), rect(7,5)).
false.
```

`add(+R1, +R2, ?RSum)` follows the idea of "adding" rectangles that was shown on the Ruby slides on operator overloading.

```
?- add(rect(3,4),rect(5,6),R).
R = rect(8, 10).

?- add(rect(3,4),rect(5,6),rect(W,H)).
W = 8,
H = 10.

?- add(rect(3,4),rect(5,6),rect(10,10)).
false.

?- X = 10, add(rect(3,4),rect(5,6),rect(X,X)).
false.
```

Assume both terms of `rect` structures are non-negative integers.

<u>If you need more than ten mostly short lines of Prolog to implement all the above, you're probably not making good use of unification.</u>

Problem 5. (3 points) `consec.pl`

Write a predicate `consec(?A, ?B, ?C)` that expresses the relationship that `A`, `B`, and `C` are consecutive integers. `A`, `B`, and `C` can be any combination of integers and uninstantiated variables.

Examples:

```
?- consec(6,B,C).
B = 7,
C = 8.

?- consec(3,4,5).
true.

?- consec(X,Y,-3).
X = -5,
Y = -4.

?- consec(X,0,Y).
```

```
X = -1,
Y = 1.

?- consec(A,2,A).
false.
```

If none of the three terms are instantiated, we see this:

```
?- consec(A,B,C).
A = 1,
B = 2,
C = 3.
```

Implementation notes

`integer/1` can be used to see if a term is an integer or uninstantiated:

```
?- integer(5).
true.

?- integer(A).
false.

?- A = 5, integer(A).
A = 5.
```

Also, note this behavior of `is/2`:

```
?- X = 2, 3 is X + 1.
X = 2.

?- X = 2, 3 is X + 10.
false.
```

My solution has four clauses.

Problem 6. (3 points) `bases.pl`

Write a predicate `bases/2` such that `bases(+Start,+End)` prints the integers from `Start` through `End` in decimal, hex, and binary. Assume that `Start` is non-negative and that `End` is greater than `Start`. Examples:

```
$ swipl bases.pl
...

?- bases(0,5).
 Decimal     Hex         Binary
     0         0              0
     1         1              1
     2         2             10
     3         3             11
     4         4            100
     5         5            101
true.

?- bases(1022,1027).
```

```
 Decimal      Hex        Binary
   1022       3FE    1111111110
   1023       3FF    1111111111
   1024       400   10000000000
   1025       401   10000000001
   1026       402   10000000010
   1027       403   10000000011
true.
```

Be sure that your predicate succeeds, showing `true`, not `false`.

Below is a predicate `fmttest/0` that shows almost exactly the specifications to use with `format/2`. However, you'll need to do `help(format/2)` and figure out how to output numbers in hex and binary.

```
?- listing(fmttest).
fmttest :-
        format('~tDecimal~t~10|~tHex~t~20|~tBinary~t~35|\n'),
        format('~t~d~6|~t~d~16|~t~d~30|\n', [10, 20, 30]).

true.

?- fmttest.
 Decimal     Hex        Binary
    10        20             30
true.
```

Problem 7. (12 points) `grid.pl`

Write a predicate `grid(+Rows,+Cols)` that prints an ASCII representation of a grid based on a specification of rows and columns in English.

Here's an example of a grid with three rows and four columns:

```
?- grid(three,four).
+--+--+--+--+
|  |  |  |  |
+--+--+--+--+
|  |  |  |  |
+--+--+--+--+
|  |  |  |  |
+--+--+--+--+
true.
```

The grid is built with plus signs, minus signs, vertical-bars ("or" bars), and spaces. Lines have no trailing whitespace.

Unless a specification is invalid, `grid` always succeeds, producing the `true` that follows the output.

Here are two more examples:

```
?- grid(three,twenty-one).
+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+
true.

?- grid(one,one).
+--+
|  |
+--+
true.
```

Widths and heights, in English, from `one` through `ninety-nine` are recognized; numbers are one word or two hyphen-separated words.

If a number is used for either dimension instead of an English specification, the user is reminded to use English:

```
?- grid(3,four).
Use English, please!
true.
```

Hint: Use `number/1` to see if a value is a number rather than a structure.

Invalid specifications produce `Huh?`:

```
?- grid(testing,this).
Huh?
true.

?- grid(one-hundred,twenty-five).      one-hundred is out of range
Huh?
true.

?- grid(---,+++).
Huh?
true.
```

Be careful not to accept invalid combinations of words representing numbers, like `ten-four`, `twenty-twenty`, and `one-fifty`; they, too, should produce the `Huh?` diagnostic. Example:

```
?- grid(ten-four,twenty-twenty).
Huh?
true.
```

`a8/grid-hint.html` shows a solution for a simplified version of this problem, a predicate `box` that prints a rectangle of asterisks. To provide a little extra challenge for those who want it, I'm not showing that code here but please don't hesitate to take a look if you're stumped by `grid`.

Note that terms like `ninety-nine`, `thirty-seven`, `fifty-two` are simply two-atom structures with the functor `'-'`. Here's a predicate that prints the terms of such a structure:

```
parts(First-Second) :-
     format('First word: ~w; second word: ~w\n', [First,Second]).

?- parts(twenty-one).
First word: twenty; second word: one
```

```
  true.
```

`a8/numbers.txt` might save you a little typing. (Think about using a keyboard macro/keystroke recorder in your editor or maybe a Ruby program to turn the text in that file into Prolog facts.)

Problem 8. (4 points) `rsg.pl`

"`rsg`" stands for "random sentence generator".

Overall, this problem has three parts:

(1) Decide what sort of "sentences" you'd like to generate.
(2) Create a predicate `rsg` that outputs a single random sentence.
(3) Create a predictate `rsg(+N)` that calls `rsg/0` `N` times.

Here's an example of an `rsg` that generates trivial English sentences:

```
?- rsg.
The boy sat.
true.

?- rsg.
A girl ran.
true.

?- rsg.
The girl spoke.
true.
```

Here's the Prolog code for the `rsg` above:

```
rsg :- article, b, noun, b, verb, write(.), !.

article :- p(0.5), w('The').
article :- p(_),   w('A').

noun :- p(0.33), w('boy').
noun :- p(0.5), w('girl').
noun :- p(_), w('dog').

verb :- p(0.4), w('ran').
verb :- p(0.66), w('sat').
verb :- p(_), w('spoke').

w(X) :- write(X).

b :- w(' ').

p(P) :- number(P), !, random(100) < P * 100.
p(_).
```

`w/1` and `b/0` are convenience predicates that let us save a little typing.

`p(X)` is a predicate that succeeds with a probability equal to `X`. For example, `p(0.5)` succeeds half the time, on average. If `p` is not called with a number, it succeeds.

Let's consider the procedure (the clauses) for `article`, which output `The` or `A` with equal probability:

```
  article :- p(0.5), w('The').
  article :- p(_), w('A').
```

The first clause succeeds half the time. If the first clause fails, which it will half the time, the second clause is tried, and it always succeeds. Effectively, its probability is also 0.5 but it's important that one always succeeds, so we'll use the convention of using `p(_)` on the last clause to stand for "otherwise". (Yes, we could just omit it, too, but having a `p` goal on each clause seems more aesthetically appealing at the moment.)

And now, a confession: I made a dopey mistake when writing this problem. Here was my first version of `noun`:

```
  noun :- p(0.33), w('boy').
  noun :- p(0.33), w('girl').
  noun :- p(_), w('dog').
```

My thinking was that I'd get an even three-way distribution between `boy`, `girl`, and `dog` but for 10,000 calls to `noun` I found that I was getting counts like these:

```
      3297 boy
      4524 dog
      2179 girl
```

What's happening is that a third of the time, the first clause succeeds and we get `boy`. In the two-thirds of the time that the first clause fails, we then pick `girl` one third of the time, which is 2/9 overall. We reach the always succeed `p(_)` case 4/9 of the time overall and produce `dog`. Oops!

My press deadline was looming and I couldn't think of a simple way to produce an even distribution with the Prolog we've seen. (In particular, without using lists and/or some higher-order predicates.) I thought about dropping this problem altogether but I like it because it gives you a chance to be creative. Thus it remains, along with this story.

To get an even three-way split, we can do this:

```
  noun :- p(0.33), w('boy').
  noun :- p(0.5), w('girl').
  noun :- p(_), w('dog').
```

One third of the time we get `boy`. In half of the remaining two-thirds, we get `girl`. In the remaining third overall, we get `dog`.

If you do the math for the `verb` procedure shown above, I believe you'll find that we should get `ran` 40% of the time, `sat` 40% of the time, and `spoke` 20% of the time.

If you want to work through the math or maybe write a Ruby method to generate `p` values that produce a particular distribution, that's fine, but if you want to just plop in some numbers and see if they produce results you like, that's fine, too!

(End of confession and emergency problem salvage operation.)

Let's use a flexible definition for "sentence" and now say that a sentence is an array of integers and nested arrays of integers. Here are some random "sentences" of that sort:

```
?- rsg.
[[17,91,30,31,38,40],85,48,96,[79,62]]
true.

?- rsg.
[61]
true.

?- rsg.
[[3,58,[1,[21,95,6,85,9,92,38,79,27,24,2,10,47,[6,[96,58,62],57,56]
],44],[83,48,14,60,79,[29,6,49,93,55,24]],2,96,23,64,69,97,[[37,32,
66,12],41,9],36,[[33,76],45],74,37,[5],72,55,86,[16,9,30],41],[[[92
,3],90,39],64,18,22,19],47,65,57,49]
true.
```

A fourth result, that's not shown above, was 23,424 characters long.

Here's the code that generated the arrays above:

```
rsg :- array, !.

array :- w('['), elems, w(']').

elems :- p(0.2), elem.
elems :- p(_), elem, w(','), elems.

elem :- p(0.8), X is random(100), w(X).
elem :- p(_), array.
```

Note the procedure for `elems`. 20% of the time it outputs a single element. 80% of the time it outputs an element (`elem`) followed by a comma and more elements (`elems`—a recursive call).

The procedure for `elem` outputs a random number between 0 and 99 on 80% of the calls, but on the remaining 20%, it outputs an array.

Note that the body for `rsg` should end with a cut, to prevent backtracking that would in turn produce some malformed results.

Possibilities for "sentence"

If you want to stick to English sentences, you might have some fun with a Mad Libs style involving popular culture or current events, especially the election season.

There are lots of possibilities in the symbolic realm, like expressions for some language you know, or even simple whole programs. Various possibilities with ASCII pictures come to mind. You might consider an HTML document to be a sentence. It's fine to have multi-line "sentences".

<u>I hope I'll be dazzled by some creativity but something as simple as English sentences with three or more fields with varying content is sufficient for full credit.</u>

I'll compile a sampling of random sentences generated by all solutions and post it on Piazza. I won't show authorship.

The approach shown above lets you be fairly creative using only material presented on slides 1-131 but you are free to implement `rsg/0` in any way you want. In particular, when we get into lists you'll see additional ways to randomly choose from several things.

Incidentally, another of the many things I learned from Ralph Griswold is that processing machine-generated input, like the random arrays above, often turns up bugs that have long been dormant while processing human-generated input.

Along with `rsg/0` you'll need to write `rsg(+N)`, which generates `N` random sentences using `rsg/0`. `N` is assumed to be an integer greater than zero. A line with three dashes is output after each sentence.

```
?- rsg(3).
[69,72]
---
[55,11,[18,83,80,57,1,54,13,57,[59],68,25],1]
---
[4,[[47,41,98,96]]]
---
true.
```

Testing note

Because of the nature of this problem there's no simple way to test `rsg/0` in an automated fashion. For this problem the tester only confirms that `rsg/1` produces the right number of "`---`" lines. Passing that simple test doesn't guarantee 75% of the points on this problem.

Problem 9. Extra Credit `observations.txt`

Submit a plain text file named `observations.txt` with...

(a) (1 point extra credit) An estimate of how long it took you to complete this assignment. To facilitate programmatic extraction of the hours from all submissions have an estimate of hours on a line by itself, more or less like one of the following three examples:

```
Hours: 6
Hours: 3-4.5
Hours: ~8
```

If you want the one-point bonus, be sure to report your total (estimated) hours on a line that starts with "`Hours:`". There must be only one "`Hours:`" line in `observations.txt`. It's fine if you care to provide per-problem times, and that data is useful to us, but report it in some form of your own invention, not with multiple lines that contain "Hours:", in either upper- or lower-case.

Other comments about the assignment are welcome, too. Was it too long, too hard, too detailed? Speak up! I appreciate all feedback, favorable or not.

(b) (1-3 points extra credit) Cite an interesting course-related observation (or observations) that you made while working on the assignment. The observation should have at least a little bit of depth. Think of me saying "Good!" as one point, "Excellent!" as two points, and "Wow!" as three points. I'm looking for quality, not quantity.

Turning in your work

Use `a8/turnin` to submit your work.

Line counts are often good for ballpark measurements of program size for many languages but they're sometimes misleading with Prolog. For example, I'll sometimes write procedures with one goal per line. With Prolog I'm going to give you a different measure of my solutions: the number of left parentheses and commas that appear. I'll use this bash script:

```
$ cat a8/plsize
for i in $*
do
    echo $i: $(tr -dc "(," < $i | wc -c)
done
```

Here's what I see as of press time, with comments stripped:

```
$ a8/plsize $(grep -v txt a8/delivs)
queries.pl: 129
altrules.pl: 9
sequence.pl: 17
rect.pl: 46
consec.pl: 24
bases.pl: 13
grid.pl: 131
rsg.pl: 39
```

Miscellaneous

Aside from **->** and **;** you can use any elements of Prolog that you desire, but the assignment is written with the intention that it can be completed easily using only the material presented on Prolog slides 1-131.

Point values of problems correspond directly to assignment points in the syllabus. For example, a 10-point problem on this assignment corresponds to 1% of your final grade in the course.

Feel free to use comments to document your code as you see fit, but note that no comments are required, and no points will be awarded for documentation itself. (In other words, no part of your score will be based on documentation.) In Prolog, `%` is comment to end of line. Comments with `/* ... */`, just like in Java, are supported, too.

Remember that late assignments are not accepted and that there are no late days; but if circumstances beyond your control interfere with your work on this assignment, there may be grounds for an extension. See the syllabus for details.

My estimate is that it will take a typical CS junior from 4 to 6 hours to complete this assignment.

Our goal is that everybody gets 100% on this assignment AND gets it done in an amount of time that is reasonable for them.

If you put six hours into this assignment and don't seem to be close to completing it, it's definitely time to touch base with us, regardless of whether you have any questions. Specifically mention that you've reached six hours. Give us a chance to speed you up!

I hate to have to mention it but keep in mind that cheaters don't get a second chance. If you give your code to somebody else and they turn it in, you'll both likely fail the class, get a permanent transcript notation stating you cheated, and maybe more. See the syllabus for the details.