CSCE 420, Assignment A6

due: Monday, May 9, 3:00pm

turn-in files by committing and pushing them to your TAMU github repository

This project gives some simple practice with programming in Prolog. You can use 'gprolog' on compute.cs.tamu.edu, or you can install gprolog or SWI-Prolog on your own machine, but the programs must still work with 'gprolog' on compute.cs.tamu.edu.

These questions are intended to show you several different paradigms or use-cases of Prolog.

Remember that Prolog works by back-chaining (e.g. putting antecedents on a goal stack), along with unification. The order of rules and facts is important; when trying to solve a predicate, it will try unifying it to facts or the head (consequent) of rules in the order that they appear in the program; and antecedents will be evaluated in left-to-right order (as if pushed onto stack in right-to-left order). Recall that 'is' can be used to compute variable binding by arithmetic, such as 'X is A+1' as an antecedent. See the documentation for arithmetical comparison operators like equal (=:=), not equal (=\=), greater than (>), and less-than-or-equal-to (=<, *not* <=). For more info, see the textbook, lecture slides, tutorial posted on the course web site, and online Prolog documentation. You might find 'trace.' and 'notrace.' to be helpful for debugging programs, which allows you to see the calls during back-chaining.

What to Turn In:

- Put all your predicates in one file called '**solutions.pl**'. We will load this and test it by calling your predicates on other test inputs.
- Put examples of the input and output for each question in a file called '**transcript.txt**'
- Check these into an **A6/** directory in your github repository.

1. Write rules in Prolog to infer various kinship relationships in terms of basic predicates like `parent(X,Y)` and `female(X)` and `male(Y)`. Input the following facts about people on The Simpsons:

```
parent(bart,homer).
parent(bart,marge).
parent(lisa,homer).
parent(lisa,marge).
parent(maggie,homer).
parent(maggie,marge).
parent(homer,abraham).
parent(herb,abraham).
parent(tod,ned).
parent(rod,ned).
parent(marge,jackie).
parent(patty,jackie).
parent(selma,jackie).

female(maggie).
female(lisa).
female(marge).
female(patty).
female(selma).
female(jackie).

male(bart).
male(homer).
male(herb).
male(burns).
male(smithers).
male(tod).
male(rod).
male(ned).
male(abraham).
```

Write rules to define the following relationships: `brother()`, `sister()`, `aunt()`, `uncle()`, `grandfather()`, `granddaughter()`, `ancestor()`, `descendant()`, and `unrelated()`. Use the convention that relation(X,Y) means "the relation of X is Y". For example, uncle(bart,herb) means the uncle of bart is herb.

Use your rules to answer the following queries (typing ';' after each solution to get the next):

```
?- brother(rod,X).
X = tod ;

?- sister(marge,X).
X = selma ;
X = patty ;

?- aunt(X,patty).
X = bart ;
X = lisa ;
X = maggie ;
```

```
?- uncle(bart,X).
X = herb ;

?- grandfather(maggie,X).
X = abraham ;

?- granddaughter(jackie,lisa).
true

?- ancestor(bart,X).
X = homer ;
X = marge ;
X = abraham ;
X = jackie ;

?- unrelated(tod,bart).
true

?- unrelated(maggie,smithers).
true

?- unrelated(maggie,selma).
false
```

2. Using the following database, write a Prolog query to find all the surgeons who live in Texas and make over $100,000/yr. You will have to add some additional data, such as about different types of surgeons, or city-state relationships.

```
occupation(joe,oral_surgeon).
occupation(sam,patent_laywer).
occupation(bill,trial_laywer).
occupation(cindy,investment_banker).
occupation(joan,civil_laywer).
occupation(len,plastic_surgeon).
occupation(lance,heart_surgeon).
occupation(frank,brain_surgeon).
occupation(charlie,plastic_surgeon).
occupation(lisa,oral_surgeon).

address(joe,houston).
address(sam,pittsburgh).
address(bill,dallas).
address(cindy,omaha).
address(joan,chicago).
address(len,college_station).
address(lance,los_angeles).
address(frank,dallas).
address(charlie,houston).
address(lisa,san_antonio).

salary(joe,50000).
salary(sam,150000).
salary(bill,200000).
salary(cindy,140000).
salary(joan,80000).
salary(len,70000).
salary(lance,650000).
salary(frank,85000).
salary(charlie,120000).
salary(lisa,190000).
```

3. Consider the following database of PROLOG facts:

```
subject(algebra,math).
subject(calculus,math).
subject(dynamics,physics).
subject(electromagnetism,physics).
subject(nuclear,physics).
subject(organic,chemistry).
subject(inorganic,chemistry).

degree(bill,phd,chemistry).
degree(john,bs,math).
degree(chuck,ms,physics).
degree(susan,phd,math).

retired(bill).
```

Write a predicate `canTeach(X,Y)` that defines which persons X can teach a class Y a given subject, which requires that they have a phd in the relevant field Z. For example, Susan can teach calculus because she has a phd in math. Note that, since Z is not mentioned in the head of the clause, it will effectively be treated like an existentially quantified variable in the antecedents, which be matched to any academic field during the back-chaining.

Show all solutions that are generated for the query `canTeach(X,Y).`

Modify the rule (call it canTeach2(X,Y)) to allow anybody with a PhD or an MS degree to teach a class related to their degree? Show all solutions.

Modify the rule (call it canTeach3(X,Y)) to exclude people who are retired. (hint: this requires *negation*). Show the solutions.

4. Prolog can be used to simulate iteration. But it is done in a very different way than in a procedural language like C++. It is done using recursion. Suppose you want to print the integers from N down to 1. You could write a predicate countdown(N), which would do some simple calculations and print using 'format' (in the antecedents) and then calls countdown(N-1), which would continue recursively until you hit a base case (countdown(0), which would be a separate fact that doesn't print anything).

```
countdown(0).
countdown(N) :- N>0,format('~w~n',[N]),M is N-1,countdown(M).
```

Write a similar rule for countup(A,B), which prints the numbers from A to B (inclusive). For convenience, it might help to add argument C, as a 'counter' (which will range between A and B, while A and B stay constant):

```
% wrapper function adds 3rd arg
countup(A,B) :- A<B,countup(A,B,<initial counter value>).

countup(A,B,C)… % what is the base case?
countup(A,B,C)… % what is the recursive case?
```

There are (at least) three ways to do this. First, you can try **switching the order** of the calls to 'format' and the recursive call in the antecedents of a rule like countdown. Second, you can try **mimicking countdown**, but print B-C. Third, you can change the recursion to **count upward** (by incrementing the count argument until it reaches B). Try implementing countup all 3 ways. Call them `countup1(A,B), countup2(A,B),` and `countup3(A,B).`

Show the result of `countup1(2,8), countup2(2,8),` and `countup3(2,8)` in the transcript.