CS560 Homework 1 (Version 1)

Question 1: Datalog Semantics

Consider the following knowledge base.

a ← b ∨ c.
b ← c ∨ d.
c.
d.

Write down all of the different interpretations. Indicate which ones are models of the knowledge base (i.e., make each clause in the knowledge base true). Is a a logical consequence of the knowledge base? Why or why not?

Question 2

Given the knowledge base

a ← b ∨ g.
b ← c ∨ d.
b ← e.
c.
d ← f.
f.

where \{a, b, c, d, e, f, g\} is the set of all atoms.

1. How many interpretations are there.
2. Give a model of the knowledge base.
3. Give an interpretation that is not a model of the knowledge base.
4. List all atoms that are logical consequences of the knowledge base.
5. List all atoms that are not logical consequences of the knowledge base.

Question 3

Exercise 2.2 from the textbook, parts a and b

Question 4

Question 2.3 from the textbook.

Question 5

This is similar to the programming question from Homework 0. Here, however, you will work with hierarchical structures as needed for function symbols (Section 2.8 of the textbook). Your algorithm will find all of the variables in a structure; it will not have to distinguish between predicate symbols, function symbols and
constants, which can each start with a lower case letter. In other words, just find the tokens that start with
an upper case letter. Hence, you do not need to distinguish whether something is the first item in a list, or
not.

We can have a structure like \( p(a, b(X), d(e, X, b(c), Y)) \).

We can represent this in Tcl's lists as:

```
set atom [list p a [list b X] [list d e X [list b c] Y]]
```
or

```
set atom {p a {b X} {d e X {b c} Y}}
```

To process the hierarchical structure, you will process it one token at a time by pulling the first item off of
the list. If the item is of length 1, then it is a token, and you can test if it is a variable or not. If it is of
greater than length 1, concatenate this item with the rest of the list (using the `concat` command). This
way you can use a simple while loop (rather than a recursive call to parse).

Here is some code to get you started.

```
proc parse {atom} {
  while {$atom != {}} {
    puts "Currently processing $atom"
    set first [lindex $atom 0]
    set rest [lrange $atom 1 end]

    # YOUR CODE HERE
  }

  # now print out all of the variables
  foreach var [array names seen] {
    puts "Variable $var"
  }
}
```

```
set atom [list p a [list b X] [list d e X [list b c] Y]]
parse $atom
```

```
vwait junk
```

Here is what your output should look like.

```
Currently processing p a {b X} {d e X {b c} Y}  
Currently processing a {b X} {d e X {b c} Y}  
Currently processing {b X} {d e X {b c} Y}  
Currently processing b X {d e X {b c} Y}  
Currently processing X {d e X {b c} Y}  
Currently processing {d e X {b c} Y}  
Currently processing d e X {b c} Y  
Currently processing e X {b c} Y  
Currently processing X {b c} Y  
Currently processing {b c} Y  
Currently processing b c Y  
Currently processing c Y  
Currently processing Y  
Variable X  
Variable Y
```

Hand in your code. Hand in the output of your code on two other examples that are even more complex
than the one given above.
Please make sure that the code you hand is properly indented to show the nesting. Editors, such as emacs, will do this automatically for you.

**Question 6: Probabilistic Propositional Top-Down Prover**

In this question, you will write a procedure `prove` to do top down proofs on knowledge bases that just have 0-ary predicates (propositional logic).

You should store the knowledge base as a list in the global variable `kb`. Each clause can be represented as `[list Head Conjunct1 Conjunct2 ...]`. Below is the encoding of the knowledge base for Exercise 2.3.

```lisp
set kb [list \\
     [list a b c] \\
     [list b d] \\
     [list b e] \\
     [list c] \\
     [list d h] \\
     [list e] \\
     [list f g b] \\
     [list g c k] \\
     [list j a b]]
```

Create a procedure called `prove` that takes as its argument the query it is supposed to prove. The query should be a list of conjuncts. The procedure `prove` should have a main loop. At each iteration, it will have its current answer cause. Pick the first conjunct in the body. Find all rules in your knowledge base that have a head that matches the first conjunct, and then randomly pick one of the them. Remove the first conjunct from the answer clause and replace it by the body of the rule that you chose. Keep going until either no rule applies, or you have proved the query. If no rule applies, print out the current answer clause and return 0.

To randomly pick an index of an element of a list `$l$`, you can do the following.

```lisp
set i [expr int([llength $matches] * rand())]
```

Note that we are resolving the non-determinism by randomly choosing one of the choices. Hence, if the first attempt does not prove the query, it might be simply because the wrong choices were made.

Create a second procedure called `multiprove` that calls `prove` up to a maximum of 100 times. Once `prove` has succeeded, stop, and return 1. Otherwise keep going. If you reach the maximum, return 0.

Run your code to prove `a` and prove `f`. Hand in your code and output from the two runs. Comment on how effective using a randomly choosing a rule as a means of resolving the non-determinism.