Question 1

Exercise 2.2 from the textbook, parts c

Question 2

Say that $a$ is a constant in our syntax and our $KB$ consists of the following.
$p(X)$.
Prove that $p(a)$ logically follows from $KB$ using a semantic proof in which you reason about interpretations, variable assignments, and models.

Question 3

Using the knowledge base from Question 2.4 from the textbook, give a top-down proof that $p(e)$ is true.
Your proof should show the same information as in Example 2.24: show the clause from the KB re-written with fresh variables, show the substitution list, and show the result of applying the resolution rule.
Note that you are free to pick which conjunct in the answer clause that you want to work on, and which clause from the knowledge base that you want to use.

Question 4

For the previous question, there is actually no wrong choice you could have made. Any valid choice for which conjunct or which clause to work on will lead to the right answer. This is rarely the case though. Add $q(f)$ to the KB. Give a top down derivation of $p(e)$ that results in an answer derivation that cannot be completed.
Does this mean the $p(e)$ is not derivable from the knowledge base? Explain your answer.

Question 5

Consider the knowledge base of Question 2.4 along with the following step in a derivation.

```
yes <- s(e,Y) ^ q(Y)
  use: q(X) <- s(X,Y) ^ q(Y)
  sub: {X/Y}
yes <- s(e,Y) ^ s(Y,Y) ^ q(Y)
```

In the above, a mistake was made in applying the resolution rule.
What mistake was made?
Will this proof end with an answer (in which the body of the answer clause is empty)? Explain why or why not.
Show what should have been done instead. Would that have resulted in an answer or not?
Question 6: Exercise 2.9 from textbook

Question 7: Exercise 2.10 from textbook

Question 8: Exercise 2.11 from textbook

For this question, you will use a bottom-up proof procedure. We will modify the bottom-up ground atom proof procedure that we discussed in the class to handle variables. The rule for adding atoms to the consequent set is as follows. If you have

- the rule \( h \leftarrow b_1 \land b_2 \land \ldots \land b_n \) in the \( KB \)
- \( b'_1, b'_2, \ldots, b'_n \) in \( C \)
- and there is a \( \sigma \) such that \( b_1 \sigma = b'_1, b_2 \sigma = b'_2, \ldots, b_n \sigma = b'_n \)

then add \( h \sigma \) into \( C \).

For each atom you add to \( C \), show what rule from KB you applied and which actsoms from \( C \) that you used, and give the substitution list (\( \sigma \)).

Question 9

Write the following in datalog. You can use the 1-ary predicates \textit{dog} and \textit{horse}, and the 2-ary predicates \textit{love} and \textit{faster}. You can also use the constants \textit{fido}, \textit{john}, \textit{mary}. You can also use the variables \( X \) and \( Y \). Make sure that each sentence that you write obeys the syntax of DataLog.

- \textit{fido} is a \textit{dog}
- \textit{john} loves \textit{mary}
- \textit{john} loves all \textit{dogs}
- \textit{horses} are faster than \textit{dogs}

Question 10: Programming: Substitutions for Datalog without Functions

In this question, you need to make a procedure called \texttt{substitute} that will take an expression and a substitution set, and apply the substitution. In order to simplify this problem, in this question, we will do this for arbitrary datalog expressions but without function symbols. Extending this to function symbols with be the topic of the next question.

Implement the substitution set as the dictionary \texttt{subs}. For each substitution \( V_i/t_i \), \( V_i \) will be the index and \( t_i \) will be the value. If you want to see if something, say \texttt{arg} has a substitution, you can use \texttt{arg in subs}.

I will be giving you some of the code for the \texttt{substitute} procedure. You will just need to fill in a few lines of code. However, in order to do that, you will need to understand my code.

I am also giving you two additional procedures, which will help you format your output so that it is easy to read. \texttt{pretty} calls \texttt{substitute} and then formats the input expression, substitution list, and answer so that it is easy to read. It uses \texttt{prettyexpr} to do this. Use the \texttt{test} procedure to format your output that you hand in. Do not change \texttt{test} nor \texttt{prettyexpr}.

The last procedure is \texttt{doall}. This procedure already has code to call \texttt{test} for the first part of Exercise 2.9.

```python
def substitute(expr, subs):
    # Just keep the first parameter in expr as is, as it is the name of the
    # of the predicate, which is not substituted
```
# YOUR SINGLE LINE OF CODE

# Now process the arguments of the predicate
# by substituting each part of expr.
for arg in expr[1:]:
    # YOUR LINES OF CODE

return newexpr

def prettyexpr(expr):
    str = expr[0]
    started = False
    for arg in expr[1:]:
        if not started:
            str += "("
            started = True
        else:
            str += ","
        if type(arg) is list:
            str += prettyexpr(arg)
        else:
            str += arg
        if started:
            str += ")"

    return str

def test(atom,subs):
    result = substitute(atom,subs)
    str = "Substitution of %s with" % prettyexpr(atom)
    for v in subs:
        str += " %s/%s" % (v, prettyexpr(subs[v]))
    print("%s
 %s" % (str, prettyexpr(result)))

doall():
    # here is a variant of Exercise 2.9 part a)

doall()

Hand in your code for the substitute procedure. Do not hand in the code for prettyexpr nor pretty. Also, hand in the output of running doall.

**Question 11: Programming: Substitutions for Datalog with Functions**

Now you will extend your code from the previous question so that it can deal with function symbols. Hence, the input expression to the substitute procedure can be arbitrarily nested. To deal with function symbols, you should change your code so that substitute makes a recursive call to itself if an argument is a function, which will be represented as a list. You can test if it is a list using the type command.

def substitute(expr,subs):
    # The following code deals with the end of the recursion,
    # where there is just a variable or a constant
    if not type(expr) is list:
        # YOUR LINES OF CODE
        # HINT: This part will NOT have any recursion in it.
# Construct the new expr, by substituting each part of expr.
# Just keep it simple, but make sure it works.
# So don’t worry treating the head of expr different from its body;
# the head will not match any of the substitution list anyways.
set newexpr = []
for arg in thing:
    # YOUR LINE OF CODE
    # HINT: Do all of the work using recursion.
    return newexpr

Make sure that substitute can be called with an atom, or with an arbitrary term, including just a variable by itself.

To further help you with this question, the functions prettyexpr and test from the previous question can be (and must be) used here.

Test your procedure on part (b) and (c) of Exercise 2.9. Do this by adding appropriate code to the doall procedure. For part (b) and (c) of Exercise 2.9, run the substitution separately on the head of the clause and on each atom in the body of the cause. Also add to doall a case showing that it works when you just pass in a single variable (rather than an atom), and that variable is being replaced by the substitution list.

Hand in your code for the substitute and doall procedures. Also, hand in the output of running doall. Do not hand in the code for prettyexpr nor pretty.