Lab 5: Review

1. Logic of Compound Statements

Exercise 1: Verify a Tautology/Contradiction

Write a function that checks if eacch of the logical expressions is a tautology (always true) or a contradiction (always false).

1. ((~p ^ q) ^ (q ^ r)) ^ ~q
2. (~p v q) v (p ^ ~q)

Exercise 2: Create a Truth Table for Biconditional

Write a function to generate a truth table for the folloing expressions.

1. p ^ ~r ↔ q v r
2. (p → (q → r)) ↔ ((p ^ q) → r)

Exercise 3: Generalized Truth Table for a Complex Expression

Write a Python function that generates a truth table for the folloing complex logical expressions. Your function should display all possible truth values of p, q, and r along with the result of the complex expression.

1. p ^ (~q → r)

Exercise 4: Verify Propositional Equivalence

Write a Python function to verify the propositional equivalence of each pair of logical expressions:

1. p ^ (q v r) ≡ (p ^ q v (p ^ r))
2. p → (q → r) ≡ (p ^ q) → r

These two expressions are logically equivalent, meaning they always have the same truth value. Your task is to confirm this by generating a truth table and checking whether the results of both expressions are always identical for all possible truth values of p, q and r.

Exercise 5: Validity of an Argument in Propositional Logic

Write a Python function that checks the validity of an argument with three premises and a conclusion using the logical form:

- Premise 1: p v q
- Premise 2: p → ~q
- Premise 3: p → r
- Conclusion: r

The argument is valid if whenever the premises are true, the conclusion is also true. Your function should evaluate the validity of the argument for all possible combinations of p, q, and r.

2. Logic of Quantified Statements

Exercise 6: Tarski's World

This is someone's java implementation of the real Tarski's World. It doesn't have all the functionality but is still fun to play with. There are formula and world files for some of the examples from the Tarski's World website. In order to load these files, you need to run the code as an application, not an applet. Download tarski.jar and type "java -jar tarski.jar" or "java -classpath tarski.jar ch.ethz.inf.rs.Tarski". Run the following tasks, capture your screen results and explain them.

1. **Aristotle:** These are the Aristotle's sentences. The goal is to build a world where the sentences are all true. aristotle.fml
2. **Maigret:** Like Aristotle's sentences, the goal is to make all of the sentences true by modifying the world. In this case, the objects are all in place, but you must assign names. Since we start getting into logical statements that the java version can't handle (it doesn't have the BackOf and FrontOf predicates), I stopped entering worlds and formulae after this. See if you can get the labelling right, even though you have to compute the truth of some of the statements yourself. maigret.fml maigret.wld

Ref: https://courses.cs.washington.edu/courses/cse590d/03sp/tarski/tarski.html

3. Introduction to Prolog

Exercise 7: Implementing a Knowledge Base with Inheritance

Objective: Define and query hierarchical relationships in a knowledge base.

1. Create a new Prolog file named animals.pl.
2. Define the following facts and rules:

   animal(mammal).

   animal(reptile).

   mammal(dog).

   mammal(cat).

   reptile(snake).

   reptile(lizard).

   is_animal(X) :- mammal(X); reptile(X).

3. Save and consult the file.
4. Execute the following queries:

   ```
   is_animal(dog).
   is_animal(snake).
   is_animal(elephant).
   ```
5. Modify the file to include **birds** and re-test the queries.
6. Explain your results.

Exercise 8: Building a Simple Expert System

Objective: Develop a rule-based expert system for diagnosing common illnesses.

1. Create a new Prolog file named diagnosis.pl.
2. Define a simple diagnostic system:

   ```
   symptom(john, fever).
   symptom(john, cough).
   symptom(jane, headache).
   symptom(jane, nausea).

   disease(flu) :- symptom(_, fever), symptom(_, cough).
   disease(migraine) :- symptom(_, headache), symptom(_, nausea).
   ```
3. Save and consult the file.
4. Execute the following queries:

   ```
   disease(flu).
   disease(migraine).
   ```
5. Modify the system to accept a **patient name** in the disease query and re-test it.
6. Explain your results.

Exercise 9: Implementing a Pathfinding Algorithm

Objective: Implement a **depth-first search (DFS)** algorithm to find paths in a graph.

1. Create a new Prolog file named graph.pl.
2. Define a directed graph using edge/2 facts:

```
edge(a, b).
edge(a, c).
edge(b, d).
edge(c, d).
edge(d, e).
```

3. Implement a recursive predicate path/2 that finds a path between two nodes:

```
path(X, Y) :- edge(X, Y).
path(X, Y) :- edge(X, Z), path(Z, Y).
```

4. Save and consult the file.
5. Execute the following queries:

```
path(a, e).
path(b, c).
```

6. Modify the program to **return the full path** as a list.
7. Explain your results.

Exercise 10: Creating a Rule-Based Chatbot

Objective: Develop a simple chatbot using Prolog's pattern-matching capabilities.

1. Create a new Prolog file named chatbot.pl.
2. Define chatbot rules based on user inputs:

```
respond(hello) :- write('Hello! How can I help you?').
respond(how_are_you) :- write('I am just a Prolog chatbot, but I am doing fine!').
respond(bye) :- write('Goodbye! Have a great day!').
respond(_) :- write('I am not sure how to respond to that.').
```

3. Save and consult the file.
4. Execute the following queries:

```
respond(hello).
respond(how_are_you).
respond(what_is_your_name).
```

5. Modify the chatbot to recognize **more responses** and support **conversational memory**.

6. Explain your results.