Resolution

In this assignment, you will gain insight into resolution by implementing a theorem prover for a subset of first order logic (FOL).

Input/Output

You will be given a single file encoding a knowledge base (KB) in conjunctive normal form (CNF), for example. The file will first list the predicates, variables, constants, and functions used in the knowledge base. It will then list the clauses on each line with whitespace separating each literal. Negation will be indicated by "!". For instance, the clauses may read:

!dog(x0) animal(x0)
dog(Kim1)

Which represents two clauses:

[¬dog(x0), animal(x0)]
[dog(Kim1)]

Which can be expressed in FOL as:

(∀ x ¬dog(x) ∨ animal(x)) ∧ dog(Kim1)

Your program should then determine if the KB is satisfiable and print either, "yes" (it is satisfiable; it cannot find any new clauses) or "no" (it is not satisfiable; it finds the empty clause). For testing purposes, it should not print anything else - please cleanup your debugging output before submitting!

Note: This differs slightly from what we typically do in class, you are not converting anything to CNF nor are you taking a query. Assume both of those steps have been done for you and the results are in the .cnf files. All you are doing is trying to derive the empty clause.

Expressiveness

You do not have to recognize all FOL expressions. Specifically, your program will not be tested using equality, e.g. "Kim01=Zori11", nor will it be required to handle nested functions. Your program will be tested on FOL at most as expressive as those found in the test cases linked at the bottom of this write up. You do not have to worry about semi-decidability, a proper implementation will halt on all of the test cases (but not on every FOL expressions in general).

Design

You must implement resolution which requires you to consider several things:

- How will you represent the following?
  - **Clauses**, which are composed of predicates
  - **Predicates**, which could be negated and have several types of arguments
  - **Terms**, which form the arguments for predicates and functions
    - **Constants**
    - **Variables**
    - **Functions**
- How will you create, store, and compare clauses?
- How will you test that if you have seen a clause before?

- How will your unification function look?
  - What must be compared?
  - How do you handle substituting a free variable for one of following?
    - Another free variable
    - A constant
    - A function
  - How do you create a new free variable to prevent name conflicts?
  - How will you apply the substitutions to predicates and functions?

It is suggested you first start by implementing a resolver (**Check out PL-Resolution in R&N 7.5 on pg 228 for ideas**) that can handle the propositional test cases before moving on to implementing unification (**Check out Unify in R&N 9.2 on pg 285 for ideas**).

I/O Format

Name your program 'lab2'. It should take 1 argument, KB.cnf.

e.g. `python3 lab2.py testcases/functions/f1.cnf`

It should then only output `yes` or `no`.

Evaluation

This lab will primarily be evaluated using test cases. Ensure that any broken code does not affect the other working components of your submission. Your score will be based on how many testcases your program passes. These will be broken up into categories with the following weights.

- Propositional Test Cases: 25%
- Constants Test Cases: 15%
- Universal Test Cases: 25%
- Universal+Constants Test Cases: 15%
- Functions Test Cases: 20%

These test cases will give you an idea of what each category involves. There are also proof sketches for each test case which might be helpful to some.

Since randomly answering "yes" or "no" will give an expected score of 50%, your program must make a good faith effort at a proper implementation in order to receive any points for that category.

Submission

Zip up your code (excluding any of the testcases we provide) and submit via ~~mycourses~~ GradeScope.