Assignment 4: Theorem Prover
Due: June 29, 2023 at 11:59PM

1 Assignment 4: Theorem Prover Specification

1.1 Overview

For this assignment, you will write a resolution refutation theorem prover for first-order logic. Given a first-order theory and query, your program should report a series of proof steps, declare the query false, or derive new conclusions forever. You should implement the "Set of Support" strategy (textbook p308) to prioritize promising resolutions.

Find the starter files and repository at `https://classroom.github.com/a/TNly_ZN0`.

1.1.1 Goals

1. Practice performing resolution steps
2. Practice performing constructing a proof sequence

1.2 Input Descriptions

Your program should read a CNF description from standard input with a form like:

```
-Human(x1) | Mortal(x1)
Human(Socrates)
Animal(x2) | Loves(x3, x2)
-Loves(x4, x5) | Loves(x5, x4)
--- negated query ---
-Mortal(Socrates)
```

Assume that all variables start with a lowercase letter, and all predicates and constants start with capital letters. The precise grammar of the CNF syntax (including spacing) you can expect for each clause is:

⟨*clause*⟩ ::= ⟨*literal*⟩ `|` ⟨*clause*⟩
| ⟨*literal*⟩

⟨*literal*⟩ ::= ⟨*predicate*⟩
| `-`⟨*predicate*⟩

⟨*predicate*⟩ ::= ⟨*capital-name*⟩`(`⟨*term-list*⟩`)`

⟨*term-list*⟩ ::= ⟨*term*⟩`,` ⟨*term-list*⟩
| ⟨*term*⟩

⟨*term*⟩ ::= ⟨*capital-constant*⟩
| ⟨*lowercase-variable*⟩

Note that there are no functions in this grammar – it is not a full description of FOL, but it is an expressive subset.

See the `cnf` directory for many examples that follow the grammar. Assume that all inputs are correctly formatted, and that variable names do not appear in multiple clauses.

1.3 Output

Your program should output three sections as follows:

1.3.1 Input Clauses

An indexed list of the input clauses (starting at 1).

1.3.2 Resolution steps

The indexed sequence of resolution steps that provide a proof if a proof exists, including the clauses they were derived from. Print `No proof exists` instead if you cannot resolve the empty clause.

1.3.3 Statistics

The number of resolutions that your prover generated. Count each successful resolution even if you did not add it to the knowledge base.

Example output for `s3.cnf` (your number of generated resolutions may not match exactly):

```
1: -Human(x1) | Mortal(x1)
2: Human(Socrates)
3: Animal(x2) | Loves(x3, x2)
4: -Loves(x4, x5) | Loves(x5, x4)
5: -Mortal(Socrates)
1 and 2 give 6: Mortal(Socrates)
5 and 6 give 7: <empty>
1646 total resolutions
```

Note the special format of the empty clause.

1.4 Execution

As in earlier assignments, you should write your code in one of two languages: Java or Python. You should name your source file that has the program's main method `TheoremProver`. One of the following options should invoke your program:

```
> python TheoremProver.py < cnf/kb1.cnf
```

(Assuming the code has been compiled)

```
> java TheoremProver < cnf/kb1.cnf
```

2 Design Ideas

There are three important parts in this assignment: parsing, unification, and resolutions. Your work from assignment 3 should be valuable in the first two parts.

2.1 Resolution

Once you can hold clauses in an organized way, it's time to try to resolve pairs of clauses in the knowledge base. Keep in mind that at least one clause from the pair should come from the set of support.

The PL-Resolution algorithm on p228 of the textbook describes how to run resolution steps to derive the empty clause or determine it's impossible (or resolution steps run forever). To perform resolution, you will need a way of comparing your clauses for equality, and comparing your literals for complements that cancel out. Only add new clauses to your knowledge base if they have at least one pair of complementary literals.

If a clause has variables, perform unification on the pair of clauses to create a new clause.

3 Submission

Submit the code for your solution along with a writeup that answers the following questions

1. Describe choices you made in your code that you feel are important. Mention any specific aspects of your implementation that might be interesting as I evaluate your program.
2. What bounds can you put on the time complexity of your program? Describe a relationship between input size and runtime based on the number of clauses and the number of literals per clause.
3. What suggestions do you have for improving this assignment?

For submission, commit/push your updates to your Github repository. I automatically have access to the repository, so if you see your updated files on Github, I have access to them too.

4 Evaluation

- +55%: Parse and store all clauses
- +15%: Attempt to resolve all pairs of clauses (using the Set of Support restrictions)
- +10%: Add resolved clauses to the knowledge base
- +10%: Print resolved clauses that lead to the empty clause
- +10%: Print correct output formatting according to specification