Assignment 3: Unification for a Theorem Prover

Due: June 15, 2023 at 11:59PM

1 Specification

1.1 Overview

For this assignment, you will write a parser and unification utility for first-order logic. In the next assignment, this utility will be used as part of a resolution refutation theorem prover.

Given a file with two FOL clauses, your program should read in the clauses and then generate all possible pairs of unified clauses. Attempt unification on every pair of literals in the two clauses. For this assignment you **won't perform any resolution steps**.

Find the starter files and repository at `https://classroom.github.com/a/Pxx4JOwn`.

1.1.1 Goals

1. Practice parsing and storing CNF clauses
2. Practice performing unification

1.2 Input Descriptions

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

```
-Human(x1) | Mortal(x1)
Human(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 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_pairs` directory for many examples that follow the grammar. Assume that all inputs are correctly formatted, and that matching variable names do not appear in multiple clauses.

1.3 Output

Your program should output as follows:

Input Clauses: an indexed list of the input clauses (starting at 1).

Unified Clauses: Print pairs of clauses if unification between two literals in them succeeds **and** those literals are complementary (one negates the other).

Statistics: The number of attempted literal unifications that your utility generated, regardless of results. Count each attempt even if you did not print it.

Example results showing the only useful unification for the example:

```
1. -Human(x1) | Mortal(x1)
2. Human(Socrates)

-Human(Socrates) | Mortal(Socrates)
Human(Socrates)

Attempted 2 unifications.
```

Example results showing the possible unifications with complementary literals for `p9.conf`. Note that the variable `x1` is just an internal name for a new unified variable – you may come up with any naming convention you would like for new variables:

```
1. -A(x, C) | B(x)
2. A(y, y) | -B(y)

-A(C, C) | B(C)
A(C, C) | -B(C)

-A(x1, C) | B(x1)
A(x1, x1) | -B(x1)

Attempted 4 unifications.
```

The order that the clause pairs are printed in is unimportant.

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_pairs/p1.cnf
```

(Assuming the code has been compiled)

```
> java TheoremProver < cnf_pairs/p1.cnf
```

2 Design Ideas

There are two important parts in this assignment: parsing and unification. You can test each part before moving onto the next part, to confirm that it is working. The ideas in the textbook will be valuable for both the unification parts.

2.1 Parsing

To parse a CNF clause given the grammar, I would suggest creating a list structure with some flexibility for variable substitution. You can build a list for each clause with functions to handle each part of the

grammar: one function might handle a string like "`-Loves(x4, x5) | Loves(x5, x4)`" to form a clause, while another function handles individual terms like "`x4`" or "`Socrates`".

A collection of these lists form a knowledge base, and new lists can be constructed by analyzing pairs of existing trees. Your program should be able to read in clauses, build an internal representation, and print them back out before you implement any analysis on them.

2.2 Unification

Chapters 7-9 of the textbook go into much more detail about the unification and resolution theory and details.

Each clause can have multiple literals. Make sure that you test every pair of literals between the two clauses for all of the possibilities for unification.

Since we are not handling functions, there is a simplified algorithm for unifying two terms:

UNIFY($t1, t2$)

```
1: if one term is a constant then
2:     if other term is a constant then
3:         return t1 = t2
4:     if other term is a variable then
5:         substitute constant for variable
6: else
7:     substitute one variable for the other
```

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 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

- +40%: Read the input and tokenize the input lines
- +25%: Fully parse the input and store clauses in a data structure
- +10%: Compare pairs of literals between the two clauses
- +5%: Perform unification between two constants
- +5%: Perform unification between a constant and a variable
- +5%: Perform unification between two variables
- +10%: Correctly print clauses from a data structure in the same format that they were input