Assignment 6: Unification
CS 730/830, Spring 2025
Due at **11pm on Wed, Mar 5**

Overview

For the next assignment (assignment 7), you will write a resolution refutation theorem prover for first-order logic. This assignment (assignment 6) is to get you started.

First, you will encode a small knowledge base and some queries, testing them with the assignment 7 reference prover. Write a KB of at least 10 first-order formulae and at least 4 queries (at least one of which is not provable). Submit a trace of the reference solution answering the queries.

Second, you will write a program that, given two sentences in first-order logic, lists all possible results of unifying them. You should check out the handout on the course web page for a description of an algorithm for unification.

Input

We provide a converter from first-order logic (without equality) to CNF, so you may assume that all your input will be in Skolemized CNF. Standard input will contain one clause per line for two lines.

Example input:

```
-Human(x1) | Mortal(x1)
Human(Socrates)
```

You may assume that all variables start with lowercase letters and that all predicates, functions, and constants are capitalized. A little more precisely, the CNF syntax you should accept is:

clause	→	literal `|` clause
	|	literal
literal	→	predicate
	|	`-` predicate
predicate	→	capitalized-name `(` term-list `)`
term-list	→	term `,` term-list
	|	term
term	→	capitalized-constant-name
	|	capitalized-function-name `(` term-list `)`
	|	lowercase-variable-name

If you are rusty on writing a parser and would like to see a simple example, there is source code in OCaml for a simple recursive-descent parser on the course web page. You can use a parser generator tool like `bison` or `antlr` if you want, but it's probably overkill.

Because this is CNF, all variables are universally quantified and variables in different clauses might coincidentally have the same names. You probably want to create unique variable names when parsing the input.

Output

There might be multiple predicates that appear in the different input clauses in opposite polarity. You need to attempt unification with every complementary pair of predicates. For every successful unification, you should write the unified pair of clauses to standard output. In the above simple example, there is only one pair of predicates appearing in opposite polarity across the two input clauses and unification succeeds, so we print one pair of unified clauses:

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

If there are multiple pairs, please separate them by a blank line.

Execution

Please use `make.sh` and `run.sh` scripts as usual.
We supply:

`wffs-to-cnf` A CNF converter. Accepts first-order logic sentences on standard input, one per line, followed by the line `--- query ---`, followed by a query (on its own line). Writes CNF to standard output, along with the negated query. The accepted syntax is:

sentence	→	predicate
	|	`(` sentence connective sentence `)`
	|	quantifier lowercase-variable-name sentence
	|	`-` sentence
predicate	→	capitalized-name `(` term-list `)`
term-list	→	term `,` term-list
	|	term
term	→	capitalized-constant-name
	|	capitalized-function-name `(` term-list `)`
	|	lowercase-variable-name
connective	→	`<->` | `->` | `&` | `|`
quantifier	→	`forall` | `exists`

Note that this means every expression with a connective needs to be surrounded by its own parentheses! (Another random limitation is that names cannot start with `v`.) The simplifier also takes a `-noquery` flag, in which case it does not take a query separator line and will not negate the last formula. Lines starting with `#` are ignored as comments.

`problems.tar.bz` sample KBs. use `tar xvfj problems.tar.bz` to extract.

`unification-validator` takes your program's name (which should be `run.sh`) as its first argument. Passes standard input to your program. Runs your program, parses its output, and verifies that it returns all (and only) possible unifications.

`unify-reference` a sample solution

Submission

Electronically submit using the instructions on the course web page, including your source code as well as a transcript of your program running with the validator. Keep your KB and queries in five separate files. Your write-up should contain traces of the reference solution running on them, as well as answers to the following questions:

1. Anything we should know about your KB and queries?
2. Describe any implementation choices you made that you felt were important. Clearly explain any aspects of your program that aren't working. Mention anything else that we should know when evaluating your work.
3. What suggestions do you have for improving this assignment in the future?

Evaluation

Roughly 2 points for the KB and 8 for the unifier.

Design Suggestions

Only try using flex and bison/antlr for your parser if you already know how to use them and want to. I implemented my CNF parser by writing a simple 'recursive-descent' parser using something like the following functions:

parse s Takes a string, returns a sentence

parse-sentence s i Takes a string and an index. Tries to parse a sentence starting from the index in the string. Returns the sentence and the index after the last character consumed.

parse-literal s i Takes a string and an index. Tries to parse a literal starting at the index in the string. Returns the literal and the index after the last character consumed.

parse-terms s i Takes a string and an index. Tries to parse a list of terms starting at the index in the string. Returns the terms and the index after the last character consumed.

parse-term s i Takes a string and an index. Tries to parse a single term starting at the index in the string. Returns the term and the index after the last character consumed.

next s i Takes a string and an index. Returns the index of the first non-whitespace character at or after the index.

token-end s i Takes a string and an index. Returns the index of the first non-name character at or after the index.

The class website includes example code for a simple recursive-descent parser in both OCaml and Python. Note that two sentences might resolve on more than one pair of complementary literals at once.