Com S 472/572 Fall 2020

Project 2: A Theorem Prover for Propositional Logic (120 pts)

Due at **5:00pm**

Monday, Nov 16

1. Introduction

In this project, you are asked to implement a theorem prover using resolution refutation for a knowledge base (KB) that consists of propositional logic (PL) sentences. The BNF grammar of PL with operator precedence is given below:

$$
\begin{array}{rcl}
\mathit{Sentence} & \rightarrow & \mathit{AtomicSentence} \mid \mathit{ComplexSentence} \\
\mathit{AtomicSentence} & \rightarrow & \textit{true} \mid \textit{false} \mid P \mid Q \mid R \mid \ldots \\
\mathit{ComplexSentence} & \rightarrow & (\ \mathit{Sentence}\ ) \\
 & \mid & \neg\ \mathit{Sentence} \\
 & \mid & \mathit{Sentence} \land \mathit{Sentence} \\
 & \mid & \mathit{Sentence} \lor \mathit{Sentence} \\
 & \mid & \mathit{Sentence} \ \Rightarrow\ \mathit{Sentence} \\
 & \mid & \mathit{Sentence} \ \Leftrightarrow\ \mathit{Sentence} \\
 & & \\
\text{OPERATOR PRECEDENCE} & : & \neg, \land, \lor, \Rightarrow, \Leftrightarrow
\end{array}
$$

Each of the five logical operators is represented by one to three special characters as shown in the following table:

¬	∧	∨	⇒	⇔
~	&&	||	=>	<=>

- All propositional symbols (i.e., atomic sentences) are strings of English letters that begin with a capital letter. For example, `Rain` and `Warm` can be atomic sentences, so can `Abcde`.
- The only exceptions to the above naming convention are the constant truth value symbols **`true`** and **`false`**, both of which begin with a lowercase letter.

- The only non-English characters are those used for representing the logical operators, the left parenthesis (, and the right parenthesis).

The input file starts with a line "Knowledge Base:" and then follows (after a blank line) with PL sentences separated by blank lines. A long PL sentence may occupy multiple lines. Consecutive lines (with no separation by a blank line) form a string that represents one PL sentence. No two consecutive blank lines appear in the input file. The following assumption can be made:

The input strings are always syntactically correct.

For example, the input line below

~(P && ~ Q) || R   =>   S && ~T

represents the sentence

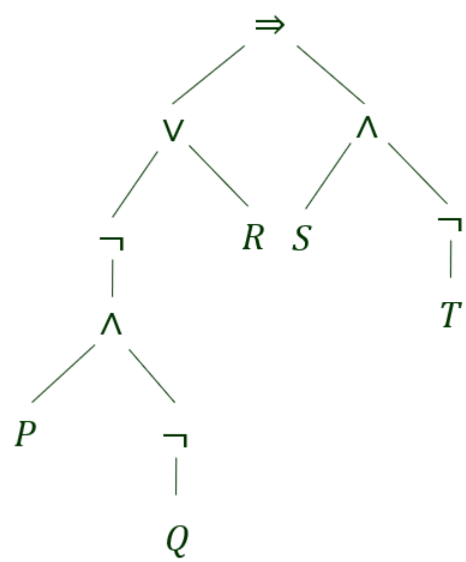
$$\neg(P \wedge \neg Q) \vee R \Rightarrow S \wedge \neg T$$

The input file ends with multiple sentences to prove using the KB. These sentences begin after a line that reads "`Prove the following sentences by refutation:`", and they are also separated by blank lines. Below is a sample input file `kb.txt`:

```
Knowledge Base:

( Rain && Outside ) => Wet

( Warm && ~Rain ) => Pleasant

~Wet

Outside

Warm


Prove the following sentences by refutation:

Pleasant

Rain
```

2. Syntax Parsing and Expression Tree Construction

Your first task is to parse every string in the input KB into atomic sentences and logical connectives. Then, construct an *expression tree* (introduced in Com S 228 on data structures). You may read about expression trees on Wikipedia (https://en.wikipedia.org/wiki/Binary_expression_tree). For example, the expression tree for the sentence $\neg(P \wedge \neg Q) \vee R \Rightarrow S \wedge \neg T$ is given below:



The construction will be similar to the algorithm used to convert an infix expression to a postfix expression (which the instructor used to teach in Com S 228). Please read the two PowerPoint files `postfix.pptx` and `infix2postfix.pptx` that he had prepared for that course to learn how the infix-to-postfix conversion works in case you were not familiar with the topic before. This conversion algorithm uses a stack. Aside from a different set of operators (with their precedence given in the table earlier), there is now a unary operator ~ that needs to be handled. Otherwise, expression tree construction pretty much resembles the infix-to-postfix conversion. Instead of outputting an operator after its two operands in the postfix format, now you just make the logical operator the parent of the two roots of the subtrees that store the same operator's subexpression operands.

You may use the following precedence table for the five logical operators and the two parentheses:

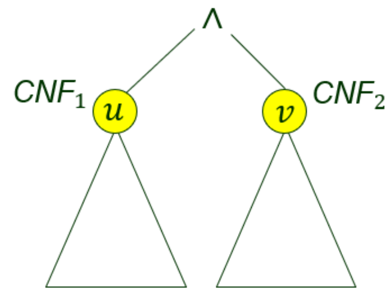
<table>
<tr><th></th><th>~</th><th>&&</th><th>||</th><th>=></th><th><=></th><th>(</th><th>)</th></tr>
<tr><td>Input precedence</td><td>5</td><td>4</td><td>3</td><td>2</td><td>1</td><td>6</td><td>0</td></tr>
<tr><td>Stack precedence</td><td>5</td><td>4</td><td>3</td><td>2</td><td>1</td><td>-1</td><td>0</td></tr>
</table>

Even though ~ is right associative, this need not be dealt with during the conversion for the following reason. If an ~ is the next character in the input string and another ~ happens to be at the top of the stack, they will cancel each other out.

3. Conversion to the Conjunctive Normal Form (CNF)

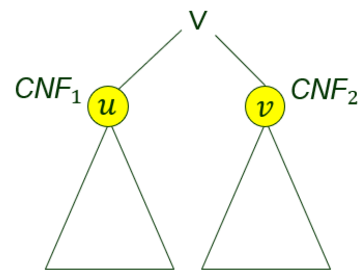
Next, you work on the expression tree of every input PL sentence in a post-order traversal to convert the sentence into a CNF. When visiting an internal node n (which represents a logical operator), its left and right children (or its unique child in the case of a ~ node that represents negation) store the CNFs for the expressions represented by the left and right subtrees. Conversion is done in a case-by-case manner depending on the logical operator stored at n. There are five cases in total:

a) The simplest case is when the node n represents a conjunction, as illustrated on the right (with the logical connective $\wedge$ shown instead of && for readability). In this case, the new CNF simply takes the form $CNF_1 \wedge CNF_2$.



b) The node n represents a disjunction $CNF_1 \vee CNF_2$, where

$$CNF_1 \equiv C_1 \wedge \cdots \wedge C_k$$
$$CNF_2 \equiv C'_1 \wedge \cdots \wedge C'_m$$



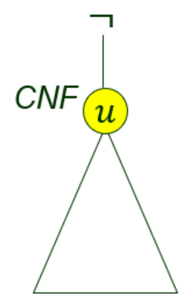
with $C_1, \ldots C_k, C'_1, \ldots, C'_m$ being clauses. Their union can be rewritten as the following CNF by repetitively distributing $\vee$ over $\wedge$. More specifically, the CNF is a conjunction of km clauses.

$$CNF_1 \vee CNF_2 \equiv \bigwedge_{\substack{i=1,\ldots,k \\ j=1,\ldots,m}} C_i \vee C'_j$$

c) The node represents a negation $\neg CNF$, where

$$CNF \equiv \left(l_{11} \vee \cdots \vee l_{1k_1}\right) \wedge \cdots \wedge \left(l_{r1} \vee \cdots \vee l_{rk_r}\right)$$

where $l_{11}, \ldots l_{1k_1}, \ldots, l_{r1}, \ldots, l_{rk_r}$ are literals. It can be verified that the negation is logically equivalent to a conjunction of a total of $k_1 \cdot \cdots \cdot k_r$ two-literal clauses:



$$\neg CNF \equiv \bigwedge_{\substack{1 \le j_1 \le k_1 \\ \vdots \\ 1 \le j_r \le k_r}} \neg l_{1j_1} \vee \cdots \vee \neg l_{rj_r}$$

If l_{ij} is a negative literal, i.e., $l_{ij} = \neg p_{ij}$, then $\neg l_{ij}$ reduces to p_{ij} because the two occurrences of $\neg$ cancel out.

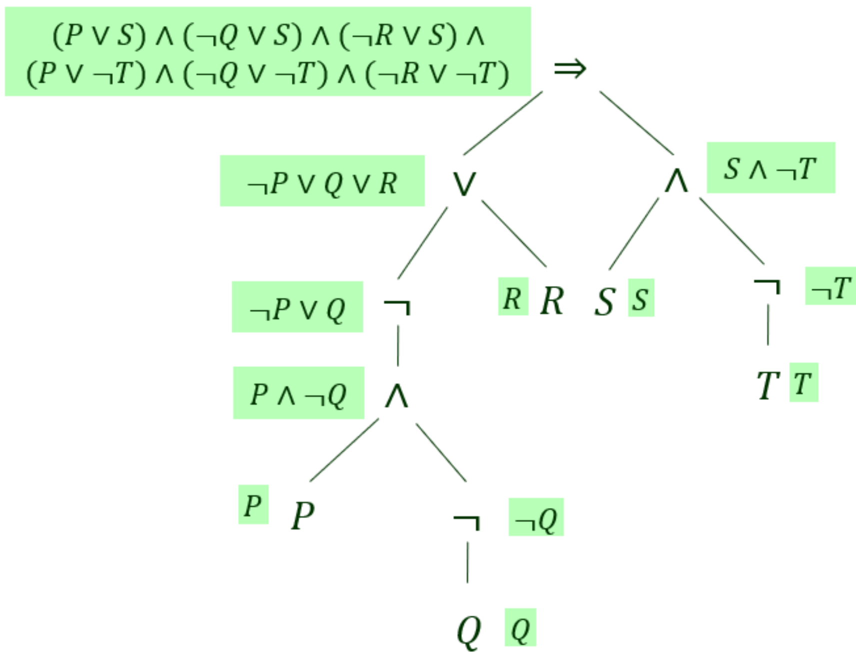
d) The node represents an implication $CNF_1 \Rightarrow CNF_2$, which is logically equivalent to $\neg CNF_1 \vee CNF_2$. First, we apply the transformation outlined in c) to convert the negation $\neg CNF_1$ into conjunctive normal form CNF'_1. Then we apply the transformation in b) to convert the disjunction $CNF'_1 \vee CNF_2$ into conjunctive normal form.

e) The node represents a biconditional sentence $CNF_1 \Leftrightarrow CNF_2$, which is logically equivalent to the conjunction of two implications:

$$(CNF_1 \Rightarrow CNF_2) \wedge (CNF_2 \Rightarrow CNF_1)$$

We apply the transformation in d) to convert $CNF_1 \Rightarrow CNF_2$ and $CNF_2 \Rightarrow CNF_1$ into conjunctive normal forms CNF_1' and CNF_2', respectively. Then simply concatenate the two forms into one as $CNF_1' \land CNF_2'$.

At each internal node n of the expression tree, you also store the CNF of the logical sentence represented by the subtree rooted at n. The expression tree example in Section 2 is redisplayed below with a CNF shown (with light green background) by the side of every node. Note that, at the left grandchild node $\neg$ of the root $\Rightarrow$, $\neg Q$ has become Q after the two $\neg$s cancel each other.



To represent a CNF, you may create three classes `ConjunctiveNormalForm`, `Clause`, and `Literal`. An object of `ConjunctiveNormalForm` is a linked list of nodes that are `Clause` objects, each of which is in turn a linked list of nodes that are `Literal` objects. A tree node n is an object of the `Node` class, whose `toString()` method is overridden to output all the clauses of the CNF stored at that node. Every clause occupies a separate line.

4. Resolution

Taking an input file, your code converts all the sentences in the KB into CNFs. These CNFs, in the same order of the original sentences, are each output as a sequence of clauses. Every clause in such a sequence occupies a separate line. A blank line separates the clause sets of every two consecutive sentences in the input file. All the clauses in the CNFs are gathered. They are all true in the *KB*.

For every propositional sentence α to prove from the input file, your system will determine whether the *KB* entails the sentence or not. This is done by first adding $\neg\alpha$ to the *KB*, and then showing that $KB \land \neg\alpha$ is unsatisfiable using resolution. You need to first convert $\neg\alpha$ into CNF, which is then split into clauses that are added to the *KB* before resolution starts.

For resolution implement the function PL-RESOLUTION (see the appendix). For efficiency, you may incorporate incremental forward chaining used in first-order logic inference. More specifically, at every iteration, resolve two clauses only if one of them was generated in the previous iteration. Your code needs to print out all the applications of the resolution rule sequentially. For every such application, you need to print out the two used clauses followed by a line “--------------------“, and then their resolvent. If no new clauses can be added and the empty clause still has not appeared, then the KB does not entail α. The following is the output generated over the input file `kb.txt` from Section 1. (You may write the output into a string and return the string.)

```
knowledge base in clauses:

~Rain || ~Outside || Wet

~Warm || Rain || Pleasant

~Wet

Outside

Warm

****************
Goal sentence 1:

Pleasant
****************

Negated goal in clauses:

~Pleasant

Proof by refutation:

~Pleasant
~Warm || Rain || Pleasant
--------------------
~Warm || Rain

~Warm || Rain
Warm
--------------------
Rain

Rain
~Rain || ~Outside || Wet
--------------------
~Outside || Wet

~Outside || Wet
Outside
```

```
--------------------
Wet

Wet
~Wet
--------------------
empty clause

The KB entails Pleasant.


****************
Goal sentence 2:

Rain
****************

Negated goal in clauses:

~Rain

Proof by refutation:

~Rain
~Warm || Rain || Pleasant
--------------------
~Warm || Pleasant

~Warm || Pleasant
Warm
--------------------
Pleasant

No new clauses are added.

The KB does not entail Rain.
```

5. Submission

Write your classes in the `edu.iastate.cs472.proj2` package. Turn in a zip file that contains the following:

a) Your source code.
b) A `README` file (optional) for comments on program execution or other things to pay attention to.

Please follow the discussion forums Project 2 Discussion and Project 2 Clarifications on Canvas. Include the Javadoc tag `@author` in each class source file. Your zip file should be named `Firstname_Lastname_proj2.zip`.

Appendix

function PL-RESOLUTION(KB, α) **returns** $true$ or $false$
  inputs: KB, the knowledge base, a sentence in propositional logic
    α, the query, a sentence in propositional logic

  $clauses \leftarrow$ the set of clauses in the CNF representation of $KB \wedge \neg\alpha$
  $new \leftarrow \{\ \}$
  while $true$ **do**
    for each pair of clauses C_i, C_j **in** $clauses$ **do**
      $resolvents \leftarrow$ PL-RESOLVE(C_i, C_j)
      if $resolvents$ contains the empty clause **then return** $true$
      $new \leftarrow new \cup resolvents$
    if $new \subseteq clauses$ **then return** $false$
    $clauses \leftarrow clauses \cup new$