Lab 06: Logical Normal Forms, Part 1

COSC 290 - Fall ’22

Starter File(s) | Lab06.zip (6 .java files)
Submission | Upload only the following file(s) to Moodle:
  • Lab06Tester.java
  • NegateProp.java
  • Prop.java
  • ConvertForm.java
Due Date | October 31st at 10:00PM for all lab sections

1 Overview

The goal of this two-part lab is to write an algorithm that converts any proposition to conjunctive normal form (CNF). It’s valuable to have propositions in CNF because we can design algorithms that are specialized for this format, such as the previous two labs, Satisfiability Solver.

2 Propositions as Trees

In this lab, we will be abstracting propositions using Binary Search Trees – a recursively defined structure. Previously, you’ve most likely seen BSTs used to store some comparable type of data (such as ints); but our proposition trees will work a bit differently. Specifically, the nodes of our tree will follow the following design:

• internal nodes (meaning nodes with at least one child) represent logical operations, such as $\lor$, $\land$, $\neg$, and $\Rightarrow$
• leaf nodes (meaning nodes with no children) represent atomic propositions – variables such as $p$, $q$, $r$, etc.

Below is an example of a tree representation of a proposition:

$$(r \Rightarrow (p \lor \neg q)) \lor \neg (q \land s)$$
Let's break down how the proposition above is abstracted into the tree:

- each non-leaf node represents some logical operator (∧, ∨, ⇒, and ¬)
- each operator (aside from ¬) has two children, representing the propositions on either side of the operator
- each leaf node represents some variable
- think about how the proposition is split amongst the branches of the tree – the root note effectively splits the original proposition into its two sub-propositions (as two sub-trees), conjoined with the ∨:
  - we have \((r ⇒ (p ∨ ¬q))\) as the left child sub-tree
  - and we have \(¬(q ∧ s)\) as the right child sub-tree

### 3 Provided Code

Provided to you are six starter files (five classes plus the tester file). A high level description of the (non-tester) classes is provided below:

- **Prop.java**: an abstract class representing all propositions. Also has some handy static function at the bottom.
- **NegateProp.java**: a proposition made up the negation of some proposition. You can visualize this as \(¬a\), where \(a\) could be a single variable, ex: \(p\), or a larger proposition, ex: \((p ∨ q ∨ r)\).
- **AtomicProp.java**: an atomic proposition (i.e. a proposition consisting of a single variable with no operator)
- **BinaryProp.java**: a proposition made up of two proposition connected by some binary logical connective (ex: \(a ∧ b\))
- **ConvertForm.java**: a collection of useful static functions for generating/transforming complex propositions; also helpful in understanding how the pieces come together.

These classes are used together to create tree abstractions of our prepositions. Relating to the example tree above, you can think of:

- **Blue** colored nodes as **BinaryProp** objects
- **Red** colored nodes as **NegateProp** objects
- **Green** colored nodes as **AtomicProp** objects
- thanks to the wonders of polymorphism, all nodes are a **Prop**

### 4 Your Task

As described previously, the end-goal of this project is to write an algorithm that can convert any proposition to CNF. You will ultimately accomplish this in three steps:

1. Given a proposition \(ϕ\), simplify \(ϕ\) so that it only contains \(∧\) and \(¬\) operators. We’ll call this "simplified" \(ϕ\).
2. Convert "simplified" \(ϕ\) into negation normal form (NNF). We’ll call this "NNF" \(ϕ\). (Note: this may introduce some ∨’s because of DeMorgan’s law)
3. Convert "NNF" \(ϕ\) into CNF. The key idea here is to distribute ∨ over ∧.

Each step takes proposition and recursively manipulates it, producing a new proposition that is logically equivalent to the input but in a desired form. In this lab, you will only implement steps 1 and 2 – step 3 will come next week.

Below is a guide to what you need to implement in this lab.

#### 4.1 Finish Helper Functions

First, spend time perusing the provided code base. Your first task is to complete all of the methods throwing “implement me” exceptions – you will find these in the Prop and NegateProp classes.
4.2 To Simplified

Next, you will implement the toSimplified() method in the ConvertForm class. This method generates a Prop that is logically equivalent to the argument, but only contains the logical connectives \(\land\) and \(\neg\). This function must be implemented recursively.

4.3 Simplified To Negation Normal Form (NNF)

Lastly, you will implement the simplifiedToNNF() method in the ConvertForm class. This function accepts a Prop that is in simplified form (as described above) and converts it to a logically equivalent Prop in NNF.

A Prop in NNF only the has negation connective (\(\neg\)) applied to atomic propositions (i.e. AtomicProp), and only contains the logical connectives \(\land\), \(\lor\), and \(\neg\). This function must be implemented recursively.

5 Pre-Lab Questions

After reading this document and provided code, answer the following before we meet (we will discuss in-lab):

1. Prop is what type of class? i.e. a(n) ____________ class.
2. Prop has three children – what are they?
3. What does getRight() return for a BinaryProp object? What about for a AtomicProp object?
4. How would you convert the following Propositions to "simplified" forms? hint: reference the table on the last page of this document:
   
   • \(\neg(q \lor s)\)
   • \(q \lor s\)
   • \(r \Rightarrow (p \lor \neg q)\)

5. The simplifiedToNNF() function assumes that the argument Prop is in "simplified" form – what does it do if it’s not (read the comments!)

6. How would you convert the following Prop to NNF forms?:
   
   • \(\neg(q \land s)\)
   • \(\neg(q \land \neg s)\)

6 Submission

See the top of this document for the lab’s due date and time. Submit only the files listed below:

- ConvertForm.java
- Lab06Tester.java
- NegateProp.java
- Prop.java

7 Tips

Below are some last tips to help you get started:

- Use paper and draw trees! Debug your algorithm by tracing through example Prop trees that you construct in Lab06Tester’s main and draw on paper.
• The `toSimplified` and `simplifiedToNNF` methods accept a `Propn` object, but really this `Prop` is the root node of some tree representing a proposition. When thinking about your recursive case(s), again apply the "black-box" method of design.

In your recursive call(s) you are passing the root node of some (sub-)tree – assuming your call works exactly as intended, what does the returned tree look like?

• Don’t forget about the provided static helper functions at `Prop.java`. Spend some time looking at the main method of `Lab06Tester` – review and experiment with the different propositions created here.

Finally, the below table is a collection of several propositional equivalences – you will make extensive use of these in this lab, so keep this table handy!

<table>
<thead>
<tr>
<th>DeMorgan’s Laws</th>
</tr>
</thead>
<tbody>
<tr>
<td>• ( \neg(p \land q) \equiv \neg p \lor \neg q )</td>
</tr>
<tr>
<td>• ( \neg(p \lor q) \equiv \neg p \land \neg q )</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Definition of Implication</th>
</tr>
</thead>
<tbody>
<tr>
<td>• ( p \Rightarrow q \equiv \neg p \lor q )</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Double Negation Law</th>
</tr>
</thead>
<tbody>
<tr>
<td>• ( \neg(\neg p) \equiv p )</td>
</tr>
</tbody>
</table>