CS 181U Applied Logic HW 1
Due Tuesday Jan 31, 2023 11:59pm

Anirudh Satish and Shaheen Cullen-Baratloo

Contents

1 Written Problems

Problem 1. In class, we discussed some challenges that come from discrepancies between the way formal logic works and the way we use logical connectives in everyday natural language. This problem will explore that a little bit more.

A. **Unless.** We use the word 'unless' all the time. For instance, I might say "I'm going on vacation *unless* I get a cold."

   i. Let the symbol U represent the 'unless' operator; we write $A\text{U}B$ to express the sentence 'A unless B'. Fill in this truth table for the unless connective for what you think the word 'unless' means in natural language. (I.e. replace the question marks with T or F.) There isn't really a right or wrong answer. I'm truly just asking you what you think, but you should be able to back up your reasoning if I were to ask.

A	B	A U B
F	F	F
F	T	T
T	F	T
T	T	F

   ii. Express the semantics that you ascribed to 'unless' using only the logical connectives from among $\wedge, \vee, \rightarrow$, and $\neg$.

   Answer.
   $A\text{U}B \equiv (A \vee B) \wedge (\neg(A \wedge B))$

iii. If you can only use connectives from among $\bar{\wedge}$ (NAND), $\bar{\vee}$ (NOR), and $\oplus$ (XOR), what is the smallest formula that you can write that is equivalent to A U B ?

Answer.

$A\text{U}B \equiv A \oplus B$

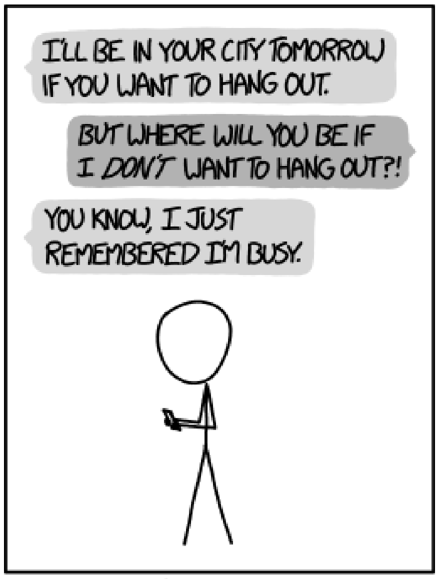
iv. Imagine that the *unless* operator, U, became a standard operator in formal propositional logic. In your opinion, do you think that the word 'unless' as understood and used in natural language and a formal semantics of the unless operator U would have any important differences? Why or why not?

Answer.

Unless would be a more intuitive way to understand the XOR operator; XOR is usually pretty unintuitive and both of us have a hard time wrapping our heads around what it means, but *unless* makes more sense to us. The one case that is a bit weird to think about is the case when both A and B are True. This situation would not really arise in real life based on the *unless* statement mentioned, so trying to get intuition for this was a bit hard. However, overall the unless operator was pretty intuitive and would not have any major differences between the natural language version and the formal logic version.

B. **More on Biscuit Conditionals.** In class we talked about so-called 'Biscuit conditionals'. We observed that when somebody says "If you are hungry, there are biscuits on the table," it doesn't really have the same meaning when we try to interpret the sentence using the semantics of the propositional logic operator $\rightarrow$.

For this question, your goal is to explain to another person (a friend, student in another course, a sibling, anybody) the discrepancy that arises in trying to interpret conditional natural language sentences using formal logic. But, importantly, try not to be a jerk about it.


I'LL BE IN YOUR CITY TOMORROW IF YOU WANT TO HANG OUT.
BUT WHERE WILL YOU BE IF I DON'T WANT TO HANG OUT?!
YOU KNOW, I JUST REMEMBERED I'M BUSY.

WHY I TRY NOT TO BE PEDANTIC ABOUT CONDITIONALS.

Write one to two paragraphs describing your experience with explaining the idea of biscuit conditionals to another person.

Answer.

Explaining everything was confusing, as there are a lot of statements that can be various states of True and False, so explaining it verbally was really difficult. We should have prepared a written truth table or something to help keep track of our thoughts. It would also have helped to walk through our thoughts before starting the explanation, as we messed up our explanation a few times and said the wrong thing (again, because there are so many Trues and Falses and cases). The person we explained it too also seemed confused, as we were in class when we first saw this statement. The truth table would have probably helped them as they attempted to understand the information dump and tried to make sense of it.

2 Coding Problems

Problem 2. A Python-based Logic Language

In this problem you will implement several fundamental operations for propositional logic.

Provided Starter Code

The file `propositional_logic.py`, which you hopefully recall from last week, contains several class definitions for defining expressions in propositional logic. For instance, to encode the formula $A \rightarrow (B \wedge \neg C)$, we would write

```
Implies(BoolVar('A'), And(BoolVar('B'), Not(BoolVar('C'))))
```

where `Implies`, `And`, `Not`, and `BoolVar` are all object constructors.

Operations on Logical Formulas

Given an object, say `F`, for a logical formula, we want to perform some useful operations on it. For instance, we may want to get the set of all variables in `F` or evaluate it under some interpretation, `I`, of those variables. To perform these operations, we will make method calls, like `F.getVars()` or `F.eval(I)`.

For this problem, we will focus on these important operations:

- `isAtom` checks if `F` is an atom.
- `isLiteral` checks if `F` is a literal.
- `getVars` returns a list of all variables in `F`.
- `isNNF` checks if `F` is in negation normal form.
- `eval` evaluates `F` under an interpretation.
- `NNF` returns the negation normal form of `F`.
- `removeImplications` returns a formula equivalent to `F` without conditionals.
- `simplify` returns a simplified version of `F`.

Provided Testing Code

Along with the starter code, you also have `test_propositional_logic.py`. There are two ways to use this file for running tests.

First, you can run the tests by loading the file and then just calling the name of the testing function. If all of the assertions in the function are true, then you will get no output, meaning that everything worked. If any of the assertions does not hold, you will get an error telling you which one.

```
In [22]: test_isLiteral()
---------------------------------------------------------------------------
AssertionError                            Traceback (most recent call last)
<ipython-input-22-b9d89e90a3b4> in <module>
----> 1 test_isLiteral()

code/test_propositional_logic.py in test_isLiteral()
     29         assert F.isLiteral()
     30         assert A.isLiteral()
---> 31         assert Not(A).isLiteral()
     32         assert Not(T).isLiteral()
     33         assert not Not(Not(T)).isLiteral()
```

The other way to run tests is to use `pytest`. You'll need to install pytest to use it. On the command line you can run the command `pytest` to run all of the tests in the testing file. The `pytest` command just looks in the current directory for files that start with `test_` and runs all tests in those files that begin with `test_`.

Initially, most of the tests will just fail since many things are not yet implemented. What will probably be more useful to you is to run a specific test function while you are developing a particular function. To do so you can, for example, run

```
pytest -k test_isNNF
```

to run just the `test_isNNF()` function.

```
>> pytest -k test_isNNF
platform linux -- Python 3.10.6, pytest-7.2.1, pluggy-1.0.0
rootdir: /Homework-01/code
collected 8 items / 7 deselected / 1 selected

test_propositional_logic.py F                                                      [

========================= FAILURES====================
_________________________ test_isNNF __________________

    def test_isNNF():
     assert T.isNNF()
     assert F.isNNF()
     assert A.isNNF()
>    assert Not(T).isNNF()
E    assert None
E     +  where None = <bound method Not.isNNF of Not(BoolConst(True))>()
E     +    where <bound method Not.isNNF of
E     +           Not(BoolConst(True))> = Not(BoolConst(True)).isNNF
E     +      where Not(BoolConst(True)) = Not(BoolConst(True))

test_propositional_logic.py:52: AssertionError
==================== short test summary info=====================
FAILED test_propositional_logic.py::test_isNNF - assert None
================== 1 failed, 7 deselected in 0.04s===============
```

Provided Usage Code

Along with the starter code and testing code, you have a file `using_propositional_logic.py`. This file shows some very basic usage of functions from `propositional_logic.py`. You may find it useful to edit this file and try things out while you are developing.

Explaining and Validating Your Solutions

I expect you to add explanations and validations of your solutions. For instance, if a question were to ask you to implement a function to convert a formula into its LaTeX-form, a good explanation and validation might look like the following:

In order to convert a formula to its LaTeXform, we add a method `tex(self)` to all relevant classes. The base cases are Boolean constants T and F and Boolean variables, which all just return their simple string format. Examples of recursive cases (show as excerpts in the code below) are disjuctions where we insert the string `‘\lor‘` between the results of recursively formatting the left and right subexpressions.

```
class BoolConst(BoolExpression):
    def tex(self):
        return self.format()
    def format(self):
        return "T" if self.val else "F"

class BoolVar(BoolExpression):
    def format(self):
        return str(self.name)
    def tex(self):
        return self.format()

class Or(BoolExpression):
    def tex(self):
        return "(" + self.exp1.tex() + " \\lor " + self.exp2.tex() + ")"
```

I validated my implementation by manually running some examples. For instance

```
f3 = Iff(And(A,B),Or(Not(B),C))
print(f3.tex())
```

which produces this output:

```
    ((A \land B) \Leftrightarrow (\neg B \lor C))
```

When pasted into a LaTeX file we get a nicely formatted result!

$((A \land B) \Leftrightarrow (\neg B \lor C))$

A. **Implement** `isLiteral`.

Use the following definitions to implement the `isLiteral` function.

- An *atom* is either a constant or a variable.
- A *literal* is either an atom or the negation of an atom.

For a propositional formula object `F`, calling `F.isLiteral()` should return `True` if and only if `F` is a literal, and `False` otherwise.

Here are several examples in python syntax:

```
BoolConst(True).isLiteral() == True
BoolConst(False).isLiteral() == True
BoolVar('A').isLiteral() == True
Not(BoolVar('A')).isLiteral() == True
Not(BoolConst(True)).isLiteral() == True
And(BoolVar('A'), BoolVar('B')).isLiteral() == False
```

The equivalent expressions in the syntax of propositional logic that are literals:

T, F, A, $\neg A$, $\neg T$

This is not a literal:

$(A \wedge B)$

Validation:

In order to check if an object is a literal, we added a method `isLiteral(self)` to all relevant classes. The simple cases are constants and variables, which just return `True` since they are literals.

```
class BoolConst(BoolExpression):
    def isLiteral(self):
        return True

class BoolVar(BoolExpression):
    def isLiteral(self):
        return True
```

For And, Or, Implies, and Iff, we didn't write an isLiteral function, which means they inherit the default one from the BoolExpression superclass.

```
class BoolExpression(object):
    def isLiteral(self):
        return False
```

The default function always returns False, which is what we want given And, Or, Implies, and Iff cannot be atoms. Finally, for Not, we return whether the thing inside is an atom or not; if it is an atom, then the Not represents a Not attached to an atom which is a literal. Otherwise, the Not isn't attached to an atom so it is not a literal.

```
class Not(BoolExpression):
    def isLiteral(self):
    return self.exp.isAtom()
```

Our implementation passed all the test cases the testing file. This tested that Atoms and the negation of Atoms returned true for isLiteral, but a double negation of an Atom and other operations such as And returned False.

B. **Implement** `getVars`.

For any formula `F`, `F.getVars()` should return a list of all unique BoolVars in the expression. there should not be duplicates.

Example: `f1 = Iff(And(A,Or(B,T)), C)`

`f1.getVars()` should return `[BoolVar(A), BoolVar(B), BoolVar(C)]`

Example: `f2 = And(T,Iff(F,T))`
`f2.getVars()` should return `[] # empty list`

Validation:

In order to get the variables from a `BoolExpression`, we added a `getVars` method to all relevant classes. The simplest cases are constants, which return nothing, and `BoolVars`, which return themselves.

```
class BoolConst(BoolExpression):
    def getVars(self):
        return []

class BoolVar(BoolExpression):
    def getVars(self):
        return [self]
```

For the rest of the classes, we wrote a recursive function that would get all the variables from the sub-expressions, and concatenate the results. The implementation of the `getVars(self)` function for `And`, `Or`, `Implies` and `Iff` are the same, since they are all two expression operators, so we only included the code snippet for one of them. We turn the result into a set and back into a list to remove duplicates.

```
class Not(BoolExpression):
    def getVars(self):
        return self.exp.getVars()

class And(BoolExpression):
    def getVars(self):
        return list(set(self.exp1.getVars() + self.exp2.getVars()))
```

We tested our implementation by making sure they passed all the tests in the test file. We noticed that the provided tests seemed to cover all the cases we wanted to check, so we did not add any additional tests. Essentially, we verified that for different kinds of operators, we got the variables that we wanted, and did not get duplicates even if the original expression had duplicates.

C. **Implement** `isNNF`.

An expression is in NNF if all negations (if there are any) are "at the lowest" level and does not contain and conditionals. That is, negations only occur on atoms. For example, all of these formulas are in negation normal form:

T, F, A, $\neg T$, $\neg A$, $(A \wedge B)$, $(A \vee T)$, $(\neg C \vee A)$, $((((B \vee \neg F) \vee (P \vee \neg T)) \vee \neg A) \wedge (\neg T \vee \neg F))$

These formulas are not in negation normal form:

$\neg\neg A$, $(A \Leftrightarrow T)$, $(\neg\neg T \wedge \neg A)$, $\neg(A \wedge B)$

NOTE: This is one place where you might want to use `isinstance(self.exp, Not)` to check for two layers of negations. If you really love OOP and want to try to attempt a solution based on 'double dispatch', be my guest!

Validation:

In order to check if a `BoolExpression` is in NNF, we added an `isNNF` method to all relevant classes. The simplest cases are constants and variables, which always are in NNF.

```
    class BoolConst(BoolExpression):
        def isNNF(self):
            return True

    class BoolVar(BoolExpression):
        def isNNF(self):
            return True
```

For `And` and `Or`, we just recursively check the two sub-expressions, and return false if either sub-expression is not in NNF.

For `Not`, we check if the sub-expression is an atom. If it is an atom, then the `Not` is in NNF, but if anything more complicated than an atom is inside, then the conditions for NNF are violated.

```
    class Not(BoolExpression):
        def isNNF(self):
            return self.exp.isAtom()

    class And(BoolExpression):
        def isNNF(self):
            return self.exp1.isNNF() and self.exp2.isNNF()

    class Or(BoolExpression):
        def isNNF(self):
            return self.exp1.isNNF() and self.exp2.isNNF()
```

For Implies and Iff, isNNF just returns False since they are both conditionals, and for an expression to be in NNF, it must not contain any conditionals.

```
    class Implies(BoolExpression):
        def isNNF(self):
            return False
```

We looked at the test cases, and concluded that they verify all the different cases that we implemented, so we just made sure we passed the test cases. The test cases also checked that conditionals, and NOTs that aren't at the lowest level returned False when the function was called.

D. **Implement** `removeImplications`.

We can use these equivalences to remove implications from a formula so that it only has $\wedge, \vee, \neg$, and atomic expressions:

$A \rightarrow B \equiv \neg A \vee B$

$A \leftrightarrow B \equiv (A \rightarrow B) \wedge (B \rightarrow A)$

Examples:

`f1 = Not(Implies(A,Or(C,B)))`

`f1.removeImplications()` returns

`Not(Or(Not(BoolVar(A)), Or(BoolVar(C), BoolVar(B))))`

`f2 = Iff(Not(BoolVar(A)), Or(BoolVar(C), BoolVar(B)))`

`f2.removeImplications()` returns

`Or(And(Not(BoolVar(A)), Or(BoolVar(C), BoolVar(B))), And(Not(Not(BoolVar(A))), Not(Or(BoolVar(C), BoolVar(B)))))`

Validation:

To remove all implications from a formula, we added a `removeImplications` method to all relevant classes. For constants and variables, there is nothing to do, and we just return `self`. We used recursion to be able to handle nested cases that needed implications removed.

For `And`s, `Or`s, and `Not`s, we just recursively remove all implications from the sub-expressions.

```
class Not(BoolExpression):
    def removeImplications(self):
        return Not(self.exp.removeImplications())

class And(BoolExpression):
    def removeImplications(self):
        return And(self.exp1.removeImplications(),self.exp2.removeImplications())
```

For `Implies` and `Iff`, we apply the equivalence given above and then recursively apply the transformation to the sub-expressions. For `Iff`, we leverage the fact that `Implies` has its own `removeImplications` to perform a recursive call.

```
class Implies(BoolExpression):
    def removeImplications(self):
        return Or(Not(self.exp1).removeImplications(),
        self.exp2.removeImplications())

class Iff(BoolExpression):
    def removeImplications(self):
        return And(Implies(self.exp1, self.exp2).removeImplications(),
        Implies(self.exp2, self.exp1).removeImplications())
```

We validated our code by using the test cases provided, and also adding to it, to test the `Iff` cases that were not dealt with in the given code.

Here are some test cases of note.

```
assert T.removeImplications() == T

f = Not(Implies(A,Or(C,B)))
f_ = f.removeImplications()
assert f_ == Not(Or(Not(A), Or(C, B)))

d = Iff(A,B)
d_ = d.removeImplications()
assert d_ == And(Or(Not(A), B), Or(Not(B), A))
```

E. **Implement** `NNF`.

An expression can be converted to NNF by first removing all implications and then recursively applying equivalences:

$$\neg\neg A \equiv A$$
$$\neg(A \wedge B) \equiv (\neg A \vee \neg B)$$
$$\neg(A \vee B) \equiv (\neg A \wedge \neg B)$$

Each equivalence provides a way to either eliminate a negation at the outer level, or to move it down from one level into a lower level sub-formula.

NOTE: This is the other place where you might want to use `isinstance(self.exp, Not)` to check for two layers of negations. If you really love OOP and want to try to attempt a solution based on 'double dispatch', be my guest!

Validation:

To convert a formula to NNF, we added an `NNF` method to all relevant classes. For constants and variables, there is nothing to do, and we just return `self`

For `And` and `Or`, first we recursively remove all implications from the expression, which will just call `removeImplications` on the two sub-expressions. Then, we take the two sub-expressions and recursively call NNF on both of them before plugging them back into the original `And` or `Or`.

```python
    class And(BoolExpression):
        def NNF(self):
            temp = self.removeImplications()
            return And(temp.exp1.NNF(), temp.exp2.NNF())

    class Or(BoolExpression):
        def NNF(self):
            temp = self.removeImplications()
            return Or(temp.exp1.NNF(), temp.exp2.NNF())
```

For the implications, we recursively remove all the implications from the expression, then just call `NNF` on whatever is left; the remaining object should be able to turn itself into `NNF`.

```python
    class Implies(BoolExpression):
        return self.removeImplications().NNF()
```

For `Not`, we remove all implications from the expression, then check if it's `NNF`. If it is, we return it, otherwise we check if the returned expression is of type `Or`, `And`, or another `Not`. If this is the case, we perform the necessary transformations for each case from the equivalences given and then again recursively call `NNF` on this new expression. This last `NNF` call will turn the sub-expression into `NNF`.

```
    class Not(BoolExpression):
        def NNF(self):
            temp = self.removeImplications()
            if temp.isNNF():
                return temp
            elif (isinstance(temp.exp, Or)):
                return And(Not(temp.exp.exp1), Not(temp.exp.exp2)).NNF()
            elif (isinstance(temp.exp, And)):
                return Or(Not(temp.exp.exp1), Not(temp.exp.exp2)).NNF()
            elif (isinstance(temp.exp, Not)):
```

To validate our code, we ran the given tests and ensured that it passed all the assertion cases. When the `NNF` function was applied to $\neg(\neg\neg T \wedge \neg A)$, we got $(\neg T \vee A)$, which is in NNF. This is an example that tests both the implementation of `NNF` in the `Or` Class and the `Not` Class. One other example is when we apply `NNF` $\neg(\neg\neg T \wedge \neg A)$ to only the inner part of this expression, yielding $\neg(T \wedge \neg A)$, and this is not an NNF as verified by our `isNNF()` function. We also tested that $\neg(A \Rightarrow T)$, when made into an NNF actually works, and this also turned out to work. There were also 21 other tests, including some that we wrote that tested for other cases, and they all passed!

F. **Implement** `eval`.

In our python implementation of Boolean logic, an interpretation is a dictionary from variables to values.

Example:

`interp = BoolVar('A') : BoolConst(True), BoolVar('B'): BoolConst(False)`

`f.eval(interp)` should evaluate `f` under the interpretation.

Example:

`And(BoolConst(False), BoolVar('A')).eval(interp)`

should return `BoolConst(False)`

Validation:

To evaluate a formula, we added a `eval` method to all relevant classes. The simplest case is a constant, where there is nothing to do. For variables, we just look up the variable in the environment and return it.

```
    class BoolVar(BoolExpression):
        def eval(self, interp):
            return interp[self]
```

For `Not`, `And`, and `Or`, we recursively evaluate the sub-expressions, then plug them back into the relevant logical connective and pass it back into a `BoolConst` constructor. Below is the implementation for the class `And`

```
    class And(BoolExpression):
        def eval(self, interp):
            return BoolConst(self.exp1.eval(interp).val and
            self.exp2.eval(interp).val)
```

For `Implies`, we take a short-cut approach— we recursively remove all implications from the expression, and evaluate the resulting expression. This should give us back a `BoolConst`, which we can just return.

```
    class Implies(BoolExpression):
        def eval(self, interp):
            return self.removeImplications().eval(interp)
```

For `Iff`, we just check if the two sub-expressions evaluate to the same thing, and return the result of that check (This code was already given).

Testing this was quite straightforward. We noticed that all the provided test cases tested all of the classes that we implemented this function in. All these tests passed. To further check with some complicated expressions, we wrote some of our own tests. We evaluated $((\neg((\neg C \Leftrightarrow (\neg A \wedge B)) \Rightarrow (T \wedge \neg C)) \wedge D) \vee F)$, which evaluates to `True`. Based on this dictionary, `interp3 = [A : F, B : F, C : T, D : T, E : F]`, and our `eval` methods agreed with this.

G. **Implement** `simplify`.

Often, it is useful to simplify a formula based on the semantics that we know. For example, we might want to simplify $A \vee A$ into just A. In this part, I am asking you to implement all of the following simplification rules.

Negations	**Disjunctions**	**Conjunctions**
$\neg T \equiv F$	$T \vee X \equiv T$	$T \wedge X \equiv X$
$\neg F \equiv T$	$X \vee T \equiv T$	$X \wedge T \equiv X$
$\neg\neg X \equiv X$	$F \vee X \equiv X$	$F \wedge X \equiv F$
	$X \vee F \equiv X$	$X \wedge F \equiv F$
	$X \vee X \equiv X$	$X \wedge X \equiv X$

Biconditionals	**Implications**
$T \leftrightarrow X \equiv X$	$T \rightarrow X \equiv X$
$X \leftrightarrow T \equiv X$	$F \rightarrow X \equiv T$
$F \leftrightarrow X \equiv \neg X$	$X \rightarrow X \equiv T$
$X \leftrightarrow F \equiv \neg X$	$X \rightarrow T \equiv X$
$X \leftrightarrow X \equiv T$	$X \rightarrow F \equiv \neg X$

Examples and testing simplify. In this part of the assignment I am asking you to come up with your own examples and tests. You should fill in at least 5 tests for the `test_simplify()` function in the `test_propositional_logic.py` file.

NOTE: This is the last place where you might want to use `isinstance(self.exp, Not)` to check for two layers of negations. If you really love OOP and want to try to attempt a solution based on 'double dispatch', be my guest!

Validation:

To simplify an expression, we added a `Simplify` method to all relevant classes. Variables and constants just return themselves.

```python
class BoolConst(BoolExpression):
    def simplify(self):
        return self

class BoolVar(BoolExpression):
    def simplify(self):
        return self
```

For every other class (`Not`, `And`, `Or`, `Implies`, `Iff`), we first recursively simplify the sub-expressions, then just manually check the various simplification rules and apply them. We've only included the snippets for `Not`, `And` and `Iff` because they all look quite similar, except for different conditions/results in the conditionals.

```python
class Not(BoolExpression):
    def simplify(self):
        temp = self.exp.simplify()
        if isinstance(temp, BoolConst):
            return BoolConst(not temp.val)
        elif isinstance(temp, Not):
            return temp.exp.simplify()
        else:
            return Not(temp)

class And(BoolExpression):
    def simplify(self):
        temp1 = self.exp1.simplify()
        temp2 = self.exp2.simplify()
        if isinstance(temp1, BoolConst):
            if temp1.val == True:
                return temp2.simplify()
            else:
                return BoolConst(False)

        elif isinstance(temp2, BoolConst):
            if temp2.val == True:
                return temp1.simplify()
            else:
                return BoolConst(False)
        elif temp1 == temp2:
            return temp1.simplify()

        else:
            return And(temp1, temp2)

class Iff(BoolExpression):
    def simplify(self):
        e1 = self.exp1.simplify()
        e2 = self.exp2.simplify()
        if e1 == BoolConst(True):
            return e2
        elif e2 == BoolConst(True):
            return e1
        elif e1 == BoolConst(False):
```

```
        return Not(e2)
    elif e2 == BoolConst(False):
        return Not(e1)
    elif e1 == e2:
        return BoolConst(True)
    else:
        return Iff(e1, e2)
```

Testing this was tedious, but straightforward. We just incorporated a test case for every simplification rule given, to check its validity. Then to also make sure that the recursive simplification was working as expected, we created a statement with combined `And` and `Or`, and the test passed as expected.

3 Appendix

Solution Source Code

This is the entire source code of my solution.

```
TABWIDTH = 2

class BoolExpression(object):
    def __init__(self):
        super()
    def __eq__(self, other):
        return isinstance(other, self.__class__) and self.__dict__ == other.__dict__
    def __ne__(self, other):
        return not self.__eq__(other)
    def __str__(self):
        return self.__class__.__name__ + "(" + ", ".join([str(v) for v in self.__dict__.va
    def __repr__(self):
        return str(self)
    def __hash__(self):
        return(hash(str(self)))
    def getVars(self):
        return []
    def eval(self, interp):
        return BoolConst(False)
    def truthTable(self):
        vars = self.getVars()
        interps = allInterpretations(vars)
        truthValues = []
        for i in interps:
            truthValues.append(self.eval(i))
        return TruthTable(vars, interps, truthValues)
    def indented(self, d):
        return ''
    def treeView(self):
        print(self.indented(0))
    def isLiteral(self):
        return False
    def isAtom(self):
        return False
    def removeImplications(self):
        return self
    def NNF(self):
        return self
    def isNNF(self):
        return False
```

```
class TruthTable(object):
    def __init__(self, vars, interps, truthValues):
        self.vars = vars
        self.interps = interps
        self.truthValues = truthValues
    def __repr__(self):
        return str(self)
    def __str__(self):
        tableString = '\n'
        for v in self.vars:
            tableString += v.name + '\t'
        tableString += '|\n'
        tableString += '---'*len(tableString) + '\n'
        for i in range(len(self.truthValues)):
            for v in self.vars:
                tableString += self.interps[i][v].format() + '\t'
            tableString += '|\t' + self.truthValues[i].format() + '\n'
        return tableString

class BoolConst(BoolExpression):
    def __init__(self, val):
        self.val = val
    def format(self):
        return "T" if self.val else "F"
    def tex(self):
        return self.format()
    def eval(self, interp):
        return self
    def NNF(self):
        return self
    def getVars(self):
        return []
    def simplify(self):
        return self
    def indented(self,d):
        return TABWIDTH*d*' ' + str(self.val)
    def removeImplications(self):
        return self
    def isLiteral(self):
        return True
    def isAtom(self):
        return True
    def isNNF(self):
        return True
```

```
class BoolVar(BoolExpression):
    def __init__(self, name):
        self.name = name
    def format(self):
        return str(self.name)
    def tex(self):
        return self.format()
    def eval(self, interp):
        return interp[self]
    def NNF(self):
        return self
    def getVars(self):
        return [self]
    def simplify(self):
        return self
    def indented(self,d):
        return TABWIDTH*d*' ' + str(self.name)
    def removeImplications(self):
        return self
    def isAtom(self):
        return True
    def isLiteral(self):
        return True
    def isNNF(self):
        return True


class Not(BoolExpression):
    def __init__(self, exp):
        self.exp = exp
    def format(self):
        return "~" + self.exp.format()
    def tex(self):
        return '\\neg ' + self.exp.tex()
    def eval(self, interp):
        return BoolConst(not self.exp.eval(interp).val)
    def NNF(self):
        temp = self.removeImplications()
        if temp.isNNF():
            return temp
        elif (isinstance(temp.exp, Or)):
            return And(Not(temp.exp.exp1), Not(temp.exp.exp2)).NNF()
        elif (isinstance(temp.exp, And)):
            return Or(Not(temp.exp.exp1), Not(temp.exp.exp2)).NNF()
        elif (isinstance(temp.exp, Not)):
            return temp.exp.exp.NNF()

    def getVars(self):
        return self.exp.getVars()
    def simplify(self):
        temp = self.exp.simplify()
        if isinstance(temp, BoolConst):
            return BoolConst(not temp.val)
        elif isinstance(temp, Not):
            return temp.exp.simplify()
        else:
            return Not(temp)

    def indented(self,d):
        return TABWIDTH*d*' ' + "Not\n" + self.exp.indented(d + 1) + "\n"
    def removeImplications(self):
        return Not(self.exp.removeImplications())

    def isLiteral(self):
        return self.exp.isAtom()
    def isNNF(self):
        return self.exp.isAtom()


class And(BoolExpression):
    def __init__(self, exp1, exp2):
        self.exp1 = exp1
        self.exp2 = exp2
    def format(self):
        return "(" + self.exp1.format() + " & " + self.exp2.format() + ")"
    def tex(self):
        return "(" + self.exp1.tex() + " \\land " + self.exp2.tex() + ")"
    def eval(self, interp):
        return BoolConst(self.exp1.eval(interp).val and self.exp2.eval(interp).val)

    def NNF(self):
        temp = self.removeImplications()
        return And(temp.exp1.NNF(), temp.exp2.NNF())

    def getVars(self):
        return list(set(self.exp1.getVars() + self.exp2.getVars()))
    def simplify(self):
        temp1 = self.exp1.simplify()
        temp2 = self.exp2.simplify()
        if isinstance(temp1, BoolConst):
            if temp1.val == True:
                return temp2.simplify()
            else:
                return BoolConst(False)

        elif isinstance(temp2, BoolConst):
            if temp2.val == True:
                return temp1.simplify()
            else:
                return BoolConst(False)
        elif temp1 == temp2:
            return temp1.simplify()

        else:
            return And(temp1, temp2)

    def indented(self,d):
        result = TABWIDTH*d*' '
        result += "And\n"
        result += self.exp1.indented(d + 1) + "\n"
        result += self.exp2.indented(d + 1)
        return result
    def removeImplications(self):
        return And(self.exp1.removeImplications(), self.exp2.removeImplications())

    def isNNF(self):
        return self.exp1.isNNF() and self.exp2.isNNF()


class Or(BoolExpression):
    def __init__(self, exp1, exp2):
        self.exp1 = exp1
        self.exp2 = exp2
    def format(self):
        return "(" + self.exp1.format() + " | " + self.exp2.format() + ")"
    def tex(self):
        return "(" + self.exp1.tex() + " \\lor " + self.exp2.tex() + ")"
    def eval(self, interp):
        return BoolConst(self.exp1.eval(interp).val or self.exp2.eval(interp).val)

    def NNF(self):
        temp = self.removeImplications()
        return Or(temp.exp1.NNF(), temp.exp2.NNF())
    def getVars(self):
        return list(set(self.exp1.getVars() + self.exp2.getVars()))

    def simplify(self):
        temp1 = self.exp1.simplify()
        temp2 = self.exp2.simplify()
        if isinstance(temp1, BoolConst):
            if temp1.val == True:
                return BoolConst(True)
            else:
                return temp2.simplify()

        elif isinstance(temp2, BoolConst):
            if temp2.val == True:
                return BoolConst(True)
            else:
                return temp1.simplify()
        elif temp1 == temp2:
            return self.exp1.simplify()

        else:
            return Or(temp1, temp2)
    def indented(self,d):
        result = TABWIDTH*d*' '
        result += "Or\n"
        result += self.exp1.indented(d + 1) + "\n"
        result += self.exp2.indented(d + 1)
        return result
    def removeImplications(self):
        return Or(self.exp1.removeImplications(), self.exp2.removeImplications())

    def isNNF(self):
        return self.exp1.isNNF() and self.exp2.isNNF()

class Implies(BoolExpression):
    def __init__(self, exp1, exp2):
        self.exp1 = exp1
        self.exp2 = exp2
    def format(self):
        return "(" + self.exp1.format() + " => " + self.exp2.format() + ")"
    def tex(self):
        return "(" + self.exp1.tex() + " \\Rightarrow " + self.exp2.tex() + ")"
    def eval(self, interp):
        return self.removeImplications().eval(interp)
    def NNF(self):
        temp = self.removeImplications()
        return temp.NNF()

    def getVars(self):
        return list(set(self.exp1.getVars() + self.exp2.getVars()))

    def simplify(self):
        e1 = self.exp1.simplify()
        e2 = self.exp2.simplify()
        if e1 == BoolConst(True):
            return e2
        elif e1 == BoolConst(False):
            return BoolConst(True)
        elif e1 == e2:
            return BoolConst(True)
        elif e2 == BoolConst(True):
            return e1
        elif e2 == BoolConst(False):
            return Not(e1)
        else:
            return Implies(e1, e2)


    def indented(self,d):
        result = TABWIDTH*d*' '
        result += "Implies\n"
        result += self.exp1.indented(d + 1) + "\n"
        result += self.exp2.indented(d + 1) + "\n"
        return result

    def removeImplications(self):
        x = Or(Not(self.exp1).removeImplications(), self.exp2.removeImplications())
        return x

    def isNNF(self):
        return False

class Iff(BoolExpression):
    def __init__(self, exp1, exp2):
        self.exp1 = exp1
        self.exp2 = exp2
    def format(self):
        return "(" + self.exp1.format() + " <=> " + self.exp2.format() + ")"
    def tex(self):
```

```python
        return "(" + self.exp1.tex() + " \\Leftrightarrow " + self.exp2.tex() + ")"
    def eval(self, interp):
        val1 = self.exp1.eval(interp)
        val2 = self.exp2.eval(interp)
        return BoolConst(val1.val == val2.val)
    def NNF(self):
        return self.removeImplications().NNF()
    def getVars(self):
        return list(set(self.exp1.getVars() + self.exp2.getVars()))
    def simplify(self):
        e1 = self.exp1.simplify()
        e2 = self.exp2.simplify()
        if e1 == BoolConst(True):
            return e2
        elif e2 == BoolConst(True):
            return e1
        elif e1 == BoolConst(False):
            return Not(e2)
        elif e2 == BoolConst(False):
            return Not(e1)
        elif e1 == e2:
            return BoolConst(True)
        else:
            return Iff(e1, e2)


    def indented(self,d):
        result = TABWIDTH*d*' '
        result += "Iff\n"
        result += self.exp1.indented(d + 1) + "\n"
        result += self.exp2.indented(d + 1)
        return result

    def removeImplications(self):
        return And(Implies(self.exp1, self.exp2).removeImplications(), Implies(self.exp2, s

    def isNNF(self):
        return False



def dictUnite(d1, d2):
    return dict(list(d1.items()) + list(d2.items()))

def dictListProduct(dl1, dl2):
    return [dictUnite(d1,d2) for d1 in dl1 for d2 in dl2]

def allInterpretations(varList):
    if varList == []:
        return [{}]
    else:
        v = varList[0]
        v_interps = [{v : BoolConst(False)}, {v : BoolConst(True)}]
        return dictListProduct(v_interps, allInterpretations(varList[1:]))
```

Testing Source Code

This is the entire source code for my tests.

```
from propositional_logic import *

T = BoolConst(True)
F = BoolConst(False)

A = BoolVar("A")
B = BoolVar("B")
C = BoolVar("C")


# helper function to test for
# duplicates in a list
def allDistinct(var_list):
    return len(var_list) == len(set(var_list))

def test_isAtom():
        assert T.isAtom()
        assert F.isAtom()
        assert A.isAtom()
        assert B.isAtom()
        assert not Not(A).isAtom()
        assert not Not(Not(A)).isAtom()
        assert not And(A,B).isAtom()
        assert not Iff(T, And(A,B)).isAtom()



def test_isLiteral():
        assert T.isLiteral()
        assert F.isLiteral()
        assert A.isLiteral()
        assert Not(A).isLiteral()
        assert Not(T).isLiteral()
        assert not Not(Not(T)).isLiteral()
        assert not And(T, F).isLiteral()

def test_getVars():
        assert T.getVars() == []  and allDistinct(T.getVars())
        assert Not(T).getVars() == [] and allDistinct(Not(T).getVars())
        assert And(T,F).getVars() == [] and allDistinct(And(T,F).getVars())
        assert A.getVars() == [A] and allDistinct(A.getVars())
        assert And(A,F).getVars() == [A] and allDistinct(And(A,F).getVars())
        # for more than one variable convert
        # list to set in case order is different
        assert set(And(A,B).getVars()) == set([A,B]) and allDistinct(And(A,B).getVars())
        assert set(Iff(And(A,Or(B,T)), C).getVars()) == set([A,B,C]) and allDistinct(Iff(A

def test_isNNF():
        assert T.isNNF()
        assert F.isNNF()
        assert A.isNNF()
        assert Not(T).isNNF()
        assert Not(A).isNNF()
        assert And(A,B).isNNF()
        assert Or(Not(C),A).isNNF()
        assert not Not(Not(A)).isNNF()
        assert not And(Not(Not(T)),Not(A)).isNNF()
        assert not Not(And(A,B)).isNNF()
        assert not Iff(A,T).isNNF()

def test_NNF():
        assert T.NNF().isNNF()
        assert A.NNF().isNNF()
        assert Not(A).NNF().isNNF()
        assert Not(T).NNF().isNNF()
        assert And(A,B).NNF().isNNF()
        assert Or(Not(C),A).NNF().isNNF()
        assert Not(Not(A)).NNF().isNNF()
        assert And(Not(Not(T)),Not(A)).NNF().isNNF()
        assert Not(And(A,B)).NNF().isNNF()

        assert Not(T.NNF()).isNNF()
        assert Not(A.NNF()).isNNF()
        assert Not(Not(Not(A)).NNF()).isNNF()

        # assert not Iff(A,T).NNF().isNNF() // this should be an NNF
        assert Iff(A,T).NNF().isNNF()
        assert not Not(Not(A).NNF()).isNNF()
        assert not Not(Not(T).NNF()).isNNF()
        assert not Not(And(A,B).NNF()).isNNF()
        assert not Not(Iff(A,T).NNF()).isNNF()
        assert not Not(Or(Not(C),A).NNF()).isNNF()
        assert not Not(And(Not(Not(T)),Not(A)).NNF()).isNNF()
        assert not Not(Not(And(A,B)).NNF()).isNNF()

        # More test
        assert  Not(Or(Not(Not(T)),Not(A))).NNF().isNNF()
        assert  Not(And(Not(C),A)).NNF().isNNF()
        assert  Implies(A,T).NNF().isNNF()

def test_removeImplications():

        assert T.removeImplications() == T
        assert A.removeImplications() == A
        assert Not(A).removeImplications() == Not(A)
        assert And(A,F).removeImplications() == And(A,F)
        assert Implies(A,B).removeImplications() == Or(Not(A),B)

        f = Not(Implies(A,Or(C,B)))
        f_ = f.removeImplications()
        assert f_ == Not(Or(Not(A), Or(C, B)))

        d = Iff(A,B)
        d_ = d.removeImplications()

        assert d_ == And(Or(Not(A), B), Or(Not(B), A))

def test_eval():
        interp_1 = {A : T, B : F}

        assert T.eval(interp_1) == T
        assert F.eval(interp_1) == F
        assert A.eval(interp_1) == T
        assert B.eval(interp_1) == F
        assert And(A,T).eval(interp_1) == T
        assert And(B,T).eval(interp_1) == F
        assert Or(A,B).eval(interp_1) == T
        assert And(A,B).eval(interp_1) == F

        interp_2 = {A : F, B : F, C : T}
        assert F == Iff(C, And(Not(A),B)).eval(interp_2)
        assert T == Iff(Not(C), And(Not(A),B)).eval(interp_2)
        assert F == Implies(Iff(Not(C), And(Not(A),B)), And(T, Not(C))).eval(interp_2)

        D = BoolVar("D")
        E = BoolVar("E")
        interp_3 = {A : F, B : F, C : T, D : T, E : F}
        assert T == Or(And(Not(Implies(Iff(Not(C), And(Not(A),B)), And(T, Not(C)))), D), F



def test_simplify():
```

```
# This is a placeholder test.
# You should write your own tests
# to make sure your simplify is working.
assert T == Or(T,T).simplify()
assert F == Or(F,F).simplify()
assert T == Or(T,F).simplify()
assert T == Or(F,T).simplify()
assert T == Or(And(T,T), Or(T,F)).simplify()
assert B == Or(B,B).simplify()

x = And(Or(A,B), Not(C))
assert x == Or(F, x).simplify()
assert A == And(A,A).simplify()
assert x == And(x,x).simplify()
assert F == And(F, x).simplify()
assert F == And(x, F).simplify()
assert T == And(Or(T,F), Or(T,T)).simplify()

assert Iff(T, x).simplify() == x
assert Iff(x, T).simplify() == x
assert Iff(F, x).simplify() == Not(x)
assert Iff(x, F).simplify() == Not(x)
assert Iff(x, x).simplify() == T
assert Implies(T, x).simplify() == x
assert Implies(F, x).simplify() == T
assert Implies(x, x).simplify() == T
assert Implies(x, T).simplify() == x
assert Implies(x, F).simplify() == Not(x)
```