CS 181U Applied Logic HW 1
Due Wednesday Feb 5, 2020

Your Name Goes Here

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 $A \mathrm{U} B$ 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	T	?
T	F	?
T	T	?

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

Answer.

$A \mathrm{U} B \equiv$ **some expression using** $\wedge, \vee, \rightarrow, \neg$

iii. If, in addition to the logical connectives from part (ii), you can also 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.

$A \mathrm{U} B \equiv$ **some expression using** $\wedge, \vee, \rightarrow, \neg, \bar{\wedge}, \bar{\vee}, \oplus$

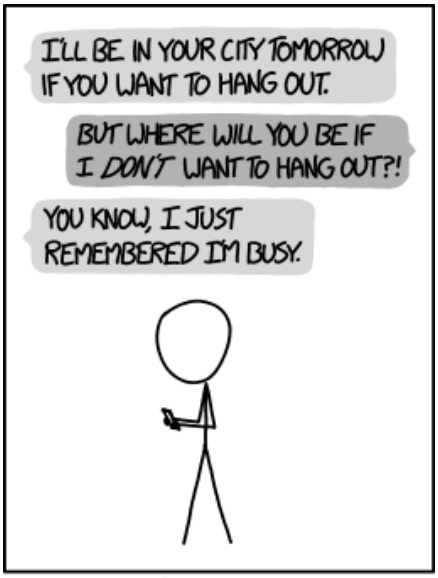
iv. Imagine that the *unless* operator, U, became a standard operator in formal propositional logic. In your opinion, do you think that the word 'unless' as understood and used in natural language and a formal semantics of the unless operator U would have any important differences? Why or why not?

Answer.

Your answer goes here.

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.



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.

Your answer goes here.

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 $A \to (B \wedge \neg C)$, 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`. On the command line, inside the starter code directory, using the CS181u Docker, 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.

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. This file will not be graded.

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:

$(A \wedge B)$

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`

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

These formulas are not in negation normal form:

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

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!

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)))))`

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!

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)`

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

$\neg T \equiv F$
$\neg F \equiv T$
$\neg\neg X \equiv X$

Disjunctions

$T \vee X \equiv T$
$X \vee T \equiv T$
$F \vee X \equiv X$
$X \vee F \equiv X$
$X \vee X \equiv X$

Conjunctions

$T \wedge X \equiv X$
$X \wedge T \equiv X$
$F \wedge X \equiv F$
$X \wedge F \equiv F$
$X \wedge X \equiv X$

Biconditionals

$T \leftrightarrow X \equiv X$
$X \leftrightarrow T \equiv X$
$F \leftrightarrow X \equiv \neg X$
$X \leftrightarrow F \equiv \neg X$
$X \leftrightarrow X \equiv T$

Implications

$T \rightarrow X \equiv X$
$F \rightarrow X \equiv T$
$X \rightarrow X \equiv T$
$X \rightarrow T \equiv X$
$X \rightarrow F \equiv \neg X$

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!