Exercise 3:

Logical Operators

In this exercise, we will define additional propositional operators to those that we have already defined in the previous exercises, and will explore the strength of various sets of such operators.

Definition. We extend the definition of a propositional formula that was given in Exercise 1 to support also the binary operators '$\rightarrow$' (*implies*, i.e., whenever the first operand is true, then so is the second operand), '$\leftrightarrow$' (*iff*, i.e., either both operands are true or both are false), '$\overline{\&}$' (*nand*, short for *not and*), and '$\bar{|}$' (*nor*, short for *not or*). The following table lists the way we denote each of these operators in Python (note that each is denoted by a sequence of two or even three characters), as well as the truth value by which the semantics of these binary operators are defined.

Operator:	$\rightarrow$	$\leftrightarrow$	$\overline{\&}$	$\bar{\|}$
Python representation:	`->`	`<->`	`-&`	`-|`
Value of '*False* operator *False*':	*True*	*True*	*True*	*True*
Value of '*False* operator *True*':	*True*	*False*	*True*	*False*
Value of '*True* operator *False*':	*False*	*False*	*True*	*False*
Value of '*True* operator *True*':	*True*	*True*	*False*	*False*

We furthermore extend the definition of a propositional formula to support also the *ternary*[1] operator '?:' (*mux*, short for *multiplexer*). More precisely, for valid formulae ϕ, ψ, and ξ, the following is also a valid formula: '$(\phi\,?\,\psi:\xi)$'. The semantics of *mux* are that the truth value of the first operand selects whether the truth value of the second or third operand is used. More precisely, if the first operand (i.e., ϕ) is true, then the truth value of the above formula is the truth value of the second operand (i.e., ψ), and otherwise that of the third operand (i.e., ξ). In prefix notation, we use '?: $\phi\ \psi\ \xi$'.

Task 1. Extend the `Formula` class from Exercise 1 to also support the four binary operators and the ternary operator defined above. For the *mux* operator, store its third operand (e.g., ξ in the above example) in a new field called `third`, and add an appropriate argument to the `__init__` method.[2] Make the necessary changes to the `infix`, `from_infix`, `prefix`, `from_prefix`, and `variables` methods. (Do not shy away from modifying the function `is_binary`, even though it was implemented for you in the skeleton file for Exercise 1.)

Task 2. Extend the `evaluate` function from Exercise 2 to also support the five operators defined above. Verify that the functions `truth_values`, `is_tautology`, and

[1]A ternary operator is an operator that takes three operands (analogous to a binary operator taking two operands, and to a unary operator taking one operand).

[2]So the additional (last) argument should be `third=None`. For the *mux* operator, `root` should be `'?:'`.

`print_truth_table` from that exercise also "inherit" the support for the above operators from your changes to the `evaluate` function.

The file `propositions/operators.py` contains a list of functions that you are asked to implement in the remainder of this exercise. We note that for any propositional formula `formula`, setting `models=all_models(formula.variables())`, and consequently calling `synthesize(models, truth_values(formula, models))`, returns a formula equivalent (in the sense of having the same truth table) to `formula`, however having only operators defined in Exercise 1. Nonetheless, using this method to reduce the set of operators is extremely wasteful in time and memory, as the number of models is exponential in the number of variables. In the next task, you are asked to implement the same functionality in a syntactic rather than semantic way, which is far more efficient.

Task 3. Implement the missing code for the function `to_not_and_or(formula)`, which takes a propositional formula and returns an equivalent formula (i.e., a formula having the same truth table) that has the same variables in it, and in addition may only have the operators *not*, *and*, and *or*, and possibly also the constants 'T' and 'F', in it. This function should run quickly even on formulae with dozens of variables. Hint: try to make local changes in order to replace each of the "illegal" operators.

A set of operators and constants[3] that is sufficient to synthesize any truth table for one or more variables is called **complete**. As you have already shown in Exercise 2, the three operators *not*, *and*, and *or* (possibly along with the constants 'T' and 'F', if you used them in your solution) constitute a complete set. As the following task shows, this set remains complete even if we remove the operator *or* from it.

Task 4 (Reduction to *not* and *and*)**.**

1. Implement the missing code for the function `to_not_and(formula)`, which takes a propositional formula and returns an equivalent formula that has the same variables in it, and in addition may only have the operators *not* and *and* (without any constants) in it. This function should run quickly even on formulae with dozens of variables. You may assume that the given formula has at least one variable in it.

2. Implement the missing code for the function `synthesize_not_and(models, values)`, which has the same behavior as the function `synthesize` that you implemented in Exercise 2, however under the additional restriction that the returned formula may only have the operators *not* and *and* (in addition to the variables from the given models, of course) in it (without any constants). Hint: do not work hard — call code that you already implemented.

We note that reducing our set of *not*, *and*, and *or* by removing the *and* operator is similarly possible, i.e., the operators *not* and *or* also constitute a complete set, and incidentally, so do the operators *not* and *implies*. While we will see in class that it is not possible to reduce any of these sets any further, i.e., that no single one of the four operators *not*, *and*, *or*, and *implies* is by itself complete, in the following tasks you will

[3]While we differentiate between operators and constants, many logic courses think of constants simply as *nullary* operators, i.e., operators taking no operands (again analogous to unary, binary, and ternary operators). For this reason, what we call "a set of operators and constants," other sources may call "a set of operators."

show two interesting facts: first, that the operator *implies* and the constant 'F' constitute together a complete set (we will see why this set is interesting in future exercises, and will only remark for now that none of *not*, *and*, *or*, or *iff* can be combined with a constant to constitute a complete set), and second, that there exist complete sets consisting only of a single operator (without any constants).

Task 5 (Reduction to *implies* and *false*)**.**

1. Implement the missing code for the function `to_implies_false(formula)`, which takes a propositional formula and returns an equivalent formula that has the same variables in it, and in addition may only have the operator *implies* and the constant 'F' in it. This function should run quickly even on formulae with dozens of variables.

2. Implement the missing code for the function `synthesize_implies_false(models, values)`, which has the same behavior as the function `synthesize`, however under the additional restriction that the returned formula may only have the operator *implies* and the constant 'F' (in addition to the variables from the given models, of course) in it.

We will now show that each of the operators *nand* and *nor* is complete, i.e., that each of these suffices by itself (without any constants) to synthesize any truth table over one or more variables. While this may seem at first glance to be an intuitive result, since the set *not* and *and* and the set *not* and *or* (which, in a sense, respectively "build up" these two operators) are each complete, this intuition is flawed. Indeed, such intuition would fail to explain why it turns out that the operator *nimplies*, short for *not implies*, is not complete even though, as noted above, the set *not* and *implies* (which "builds up" this operator in precisely the same sense) is complete. (Indeed, how would you synthesize a *not* using only *nimplies*s?)

Task 6 (Reduction to *nand*)**.**

1. Implement the missing code for the function `to_nand(formula)`, which takes a propositional formula and returns an equivalent formula that has the same variables in it, and in addition may only have the operator *nand* (without any constants) in it. This function should run quickly even on formulae with dozens of variables. You may assume that the given formula has at least one variable in it.

2. Implement the missing code for the function `synthesize_nand(models,values)`, which has the same behavior as the function `synthesize`, however under the additional restriction that the returned formula may only have the operator *nand* (in addition to the variables from the given models, of course) in it (without any constants).

Task 7 (Reduction to *nor*)**.**

1. Implement the missing code for the function `to_nor(formula)`, which takes a propositional formula and returns an equivalent formula that has the same variables in it, and in addition may only have the operator *nor* (without any constants) in it. This function should run quickly even on formulae with dozens of variables. You may assume that the given formula has at least one variable in it.

2. Implement the missing code for the function `synthesize_nor(models,values)`, which has the same behavior as the function `synthesize`, however under the additional restriction that the returned formula may only have the operator *nor* (in addition to the variables from the given models, of course) in it (without any constants).

As it turns out, *nand* and *nor* are the only complete binary operators, i.e., no other binary operator suffices by itself (without any constants) to synthesize any truth table. We now show that *mux*, when coupled with both constants, is also complete, and that this is a very convenient complete set, in a very tangible sense.

Task 8 (Reduction to *mux*)**.**

1. Implement the missing code for the function `to_mux(formula)`, which takes a propositional formula and returns an equivalent formula that has the same variables in it, and in addition may only have the operator *mux* and the constants 'T' and 'F' in it. If the given formula has no operators except *not*, *implies*, *and*, and *or*, then the number of operators in the returned formula should not be greater than that in the given formula. This function should run quickly even on formulae with dozens of variables.

2. Implement the missing code for the function `synthesize_mux(models,values)`, which has the same behavior as the function `synthesize`, however under the additional restriction that the returned formula may only have the operator *mux* and the constants 'T' and 'F' (in addition to the variables from the given models, of course) in it. The number of operators in the returned formula should not be greater than that in the formula returned by calling `synthesize` with the same parameters.

As we have seen, there are quite a few very small complete sets of operators, some of them even of size one. In fact, many logic courses take the approach of defining only a very small complete set of operators, and thinking of all other operators as shortcuts (much like in your implementation of the `to_***(formula)` functions above) for expressions written only using these operators. In doing so, however, some care is needed: e.g., how would you rewrite the formula 'F' using only *nand*s (with no variables)? In fact, while, as we have shown, *nand* and *nor* can be used to synthesize any truth table over one or more variables, none of them can be used to synthesize a truth table over no variables (i.e., a constant operator), so an additional constant is needed. Since we need "a minimum" of one operator and one constant anyway, for reasons that will become apparent in Exercises 5 and 6, it is in fact convenient (and many logic courses indeed do so) to use *implies* and 'F' (rather than *nand* and a constant, or alternatively *nor* and a constant) as the basic building blocks for propositional formulae. That being said, in the next exercises we will continue to work with a richer set of operators consisting of *not*, *implies*, *and*, and *or*, along with both constants,[4] as this set will be much more convenient to reason with. (Just think about how one represents an *or* using only *nand*s, or alternatively about how much more challenging implementing `synthesize_nand` would have been without thinking in terms of a larger set of operators.) Our choice of the set of operators to use when defining a propositional formula was therefore guided by an attempt to balance the size of the set of operators with the convenience of its use.

[4]Do not remove the functionality of *nand*, *nor*, and *mux* that you added in this exercise from your code, though!