This material will be published by Cambridge University Press as:
Mathematical Logic through Python by **Yannai A. Gonczarowski and Noam Nisan**
This pre-publication version is free to view and download for personal use only. Not for re-distribution, re-sale or use in derivative works. **Please link to: `www.LogicThruPython.org`**
© Yannai A. Gonczarowski and Noam Nisan 2017–2021.

Chapter 2:

Propositional Logic Semantics

In the previous chapter we defined the **syntax** of propositional formulas, that is, we defined which strings constitute valid propositional formulas. We were however careful not to assign any meaning, any **semantics**, to propositional formulas. The notion of the semantics of propositional formulas may be somewhat difficult to grasp, as the meaning of formulas may seem "obvious" but its formal definition may at first seem elusive. This chapter provides this formal definition.

Our intention for these semantics is as follows: every variable name (e.g., 'p' or 'q76') will stand for a certain **primitive proposition** that may be *either true or false*, independently of other primitive propositions. A compound formula that contains more than one variable name will describe a more complex proposition, whose truth or lack thereof depends on *which primitive propositions are true and which are not*. For example, we may have 'p' represent "It is raining," 'q' represent "My umbrella is open," 'r' represent "I am singing," and 's' represent "I am dancing." A compound formula like '((p&~q)&(r|s))' evaluates to *true* if it is raining and my umbrella is *not* open, and furthermore either I am singing or I am dancing.

Before moving forward with the formal definition of the semantics of propositional formulas, it may perhaps be instructive to take a short detour to another domain where we have a distinction between syntax and semantics, a domain where we expect many of our readers to have a good feel for semantics: programming languages.

1 Detour: Semantics of Programming Languages

Consider the following valid program:

```
#include <stdio.h> /*
print("wonderland")
""" */
int main() { printf("looking-glass\n"); }
// """
```

What would this program output? Well, since in Python comments start with the symbol `#` and continue until the end of the line, and multi-line strings (which are ignored on their own) are enclosed between triple quotations, then graying out Python comments and ignored strings, the above program would be interpreted as follows:

mystery_program.py

```
#include <stdio.h> /*
print("wonderland")
""" */
int main() { printf("looking-glass\n"); }
// """
```

and when executed would simply print `wonderland`. This answer, as we will now explain, while partially correct, does make some assumptions. As it turns out, the proper answer to the question of what would the above program print, is "it depends on which language you consider this program to be written in." Indeed, this program is not only a valid program in Python, but also in the C programming language![1] While the *syntax* of the above program is valid both in Python and in C, its *semantics* in each of these programming languages turn out however to be completely different. In C, comments are either enclosed between `/*` and `*/`, or start with `//` and continue until the end of the line. Therefore, as a C program, graying out C comments, the above program would be interpreted as follows:

mystery_program.c

```
#include <stdio.h> /*
print("wonderland")
""" */
int main() { printf("looking-glass\n"); }
// """
```

and when compiled and executed, would simply print `looking-glass`.

So what is our point with this example? First, that the semantics are very important: in the case of programming languages they determine what the program *does*. Second, that even if usually a short glance suffices for you to "more-or-less understand" a piece of code (or a formula), carefully defining the "right" semantics is still very important, and may be tricky and not at all obvious. With this appreciation, let us return to propositional formulas and proceed to assign *semantics* to them.

2 Models and Truth Values

Formally, the **semantics** we will give to a formula are the respective truth values that it gets in every possible setting of its variable names. We view a possible setting of these variable names as a "possible world," and the semantics of a formula are whether it is true or not in each of these possible worlds. We will call such a possible world a **model**:

Definition (Model)**.** Let S be the set of variable names. A **model** M over S is a function that assigns a truth value to every variable name in S. That is, $M : S \to \{True, False\}$.

The file `propositions/semantics.py`, which contains all of the functions that you are asked to implement in the next few sections, deals with the semantics of propositional formulas. A formula is represented as an instance of the class `Formula` that was defined in Chapter 1. We will represent a model as a Python `dict` (dictionary) that maps every variable name name to a Boolean value:

propositions/semantics.py

```
#: A model for propositional-logic formulas, a mapping from variable names to
#: truth values.
Model = Mapping[str, bool]

def is_model(model: Model) -> bool:
    """Checks if the given dictionary is a model over some set of variable
    names.
```

[1] Fear not if you have no familiarity with the C programming language. We will explain the little that is needed to know about C in order to drive the point of this discussion home.

```
    Parameters:
        model: dictionary to check.

    Returns:
        ``True`` if the given dictionary is a model over some set of variable
        names, ``False`` otherwise.
    """
    for key in model:
        if not is_variable(key):
            return False
    return True

def variables(model: Model) -> AbstractSet[str]:
    """Finds all variable names over which the given model is defined.

    Parameters:
        model: model to check.

    Returns:
        A set of all variable names over which the given model is defined.
    """
    assert is_model(model)
    return model.keys()
```

Having defined a model, we can now give each formula its semantics—the truth value that it gets in every possible model:

Definition (Truth Value of Formula in Model)**.** Given a formula ϕ and a model M over a set of variable names that contains (at least) all those used in ϕ, we define the **(truth) value** of the formula ϕ in the model M recursively in the natural way:

- If ϕ is the constant 'T', its value is *True*; if ϕ is the constant 'F', its value is *False*.
- If ϕ is a variable name p, then its value is as specified by the model: $M(p)$.
- If ϕ is of the form '$\sim\psi$', then its value is *True* if the (recursively defined) value of ψ in M is *False* (and is *False* otherwise).
- If ϕ is of the form '$(\psi\&\xi)$', then its value is *True* if the (recursively defined) values of both ψ and ξ in M are *True* (and is *False* otherwise); if ϕ is of the form '$(\psi|\xi)$', then its value is *True* if the (recursively defined) value of either ψ or ξ (or both) in M is *True* (and is *False* otherwise); if ϕ is of the form '$(\psi\rightarrow\xi)$', then its value is *True* if either the (recursively defined) value of ψ in M is *False* or the (recursively defined) value of ξ in M is *True* (and is *False* otherwise).

Returning to the example we started with, one possible model M is M('p') = *True* (it is raining), M('q') = *False* (my umbrella is NOT open), M('r') = *True* (I am singing), and M('s') = *False* (I am NOT dancing), and in this model the formula '((p&~q)&(r|s))' evaluates to the value *True* as defined recursively: '~q' evaluates to *True* (since 'q' evaluates to *False*), and so '(p&~q)' evaluates to *True* (since both 'p' and '~q' evaluate to *True*); furthermore, '(r|s)' evaluates to *True* (since 'r' evaluates to *True*); and finally '((p&~q)&(r|s))' evaluates to *True* (since both '(p&~q)' and '(r|s)' evaluate to *True*). Of course there are more possible models, and for some of them the formula evaluates to *True* while for the others it evaluates to *False*.

While the semantics of the *not* ('~'), *or* ('|'), and *and* ('&') operators are quite natural and self explanatory, the *implies* ('→') operator may seem a bit more cryptic. The way to think about '$(\psi \to \xi)$' is as stating that if ψ is *True*, then ξ is *True* as well. This statement would be *False* only if both ψ were *False* and ξ were *True*, so this statement is *True* whenever either ψ is *True* or ξ is *False*, which coincides with the above definition. Yet, it may still intuitively seem unnatural that the statement "if ψ is *True*, then ξ is *True* as well" is considered to be *True* if ψ is *False* (indeed, how should one interpret this conditional if ψ is false)? The reason for this definition is that we would generally be interested in whether a given formula is *True* in each of a *set* of models. In this context, the formula '$(\psi \to \xi)$' can be naturally interpreted as "whenever ψ is *True*, so is ξ", that is, in any model in this set in which ψ is *True*, so is ξ. (This of course still does not tell us anything about models in this set in which ψ is *False*, but replacing "if" with "whenever" may somewhat further motivate this definition, and help make this operator a bit less cryptic.)

Task 1. Implement the missing code for the function `evaluate(formula, model)`, which returns the truth value of the given formula in the given model.

propositions/semantics.py

```
def evaluate(formula: Formula, model: Model) -> bool:
    """Calculates the truth value of the given formula in the given model.

    Parameters:
        formula: formula to calculate the truth value of.
        model: model over (possibly a superset of) the variable names of the
            given formula, to calculate the truth value in.

    Returns:
        The truth value of the given formula in the given model.

    Examples:
        >>> evaluate(Formula.parse('~(p&q76)'), {'p': True, 'q76': False})
        True

        >>> evaluate(Formula.parse('~(p&q76)'), {'p': True, 'q76': True})
        False
    """
    assert is_model(model)
    assert formula.variables().issubset(variables(model))
    # Task 2.1
```

3 Truth Tables

Once we have defined the value that a formula gets in a given model, we now turn to handling sets of possible models. If we have a set of n variable names, then there are exactly 2^n possible models over this set: all possible combinations where each of the variable names is mapped to either *True* or *False*. In the next task we ask you to list all these possible models.

Before jumping to the task, we should explicitly note the exponential jump in the size of the objects that we are dealing with. While all the code that you have written so far would have no problem dealing with formulas with millions of variable names, once we want to list *all* the possible models over a given set of variable names, we will not be able

to handle more than a few dozen variable names at most: already with 40 variable names we have more than a trillion models ($2^{40} \approx 10^{12}$).

Task 2. Implement the missing code for the function `all_models(variables)`, which returns a list[2] of all possible models over the given variable names.

propositions/semantics.py

```
def all_models(variables: Sequence[str]) -> Iterable[Model]:
    """Calculates all possible models over the given variable names.

    Parameters:
        variables: variable names over which to calculate the models.

    Returns:
        An iterable over all possible models over the given variable names. The
        order of the models is lexicographic according to the order of the given
        variable names, where False precedes True.

    Examples:
        >>> list(all_models(['p', 'q']))
        [{'p': False, 'q': False}, {'p': False, 'q': True},
         {'p': True, 'q': False}, {'p': True, 'q': True}]

        >>> list(all_models(['q', 'p']))
        [{'q': False, 'p': False}, {'q': False, 'p': True},
         {'q': True, 'p': False}, {'q': True, 'p': True}]
    """
    for v in variables:
        assert is_variable(v)
    # Task 2.2
```

Guidelines: The standard term "lexicographic order" that specifies the order of the models refers to considering each model as a "word" in the alphabet consisting of the two "letters" *False* and *True*, considering the "letter" *False* to precede the "letter" *True*, and listing all the "words" (models) "alphabetically" in the sense that every word that starts with *False* precedes every word that starts with *True*, and more generally for any prefix of a "word," words that start with that prefix and then *False* (regardless of which "letters" follow) precede words that start with that prefix and then *True* (regardless of which "letters" follow).

Hint: The `product` method (with its `repeat` argument) from the standard Python `itertools` module may be useful here.

Task 3. Implement the missing code for the function `truth_values(formula, models)`, which returns a list of the respective truth values of the given formula in the given models.[3]

[2] While in this book we will not pay much attention to complexities and running times, we do pay here just a bit of attention to this first exponential blowup. Even though for simplicity the task asks to return a `list`, we recommend that readers familiar with Python *iterables* return an iterable that *iterates* over all possible models (which does not require keeping them all together in memory) rather than actually return a `list` of all possible models (which would require them to all be together in memory). The test that we provide allows for any iterable, and not merely a `list`, to be returned by this function. We do not, however, intend to run any of this code on more than a few variable names, so we do not impose any efficiency requirements on your code, and we do allow solving this task by returning a `list` of all models.

[3] Readers who implemented Task 2 to return a memory-efficient iterable rather than a `list` are encouraged to implement this method to accept `models` also as an arbitrary iterable, and to also return a memory-efficient iterable rather than a `list` from this function. The test that we provide allows for any iterable to be returned by this function, but only requires the function to support taking a `list` of models.

propositions/semantics.py

```
def truth_values(formula: Formula, models: Iterable[Model]) -> Iterable[bool]:
    """Calculates the truth value of the given formula in each of the given
    models.

    Parameters:
        formula: formula to calculate the truth value of.
        models: iterable over models to calculate the truth value in.

    Returns:
        An iterable over the respective truth values of the given formula in
        each of the given models, in the order of the given models.

    Examples:
        >>> list(truth_values(Formula.parse('~(p&q76)'),
        ...                   all_models(['p', 'q76'])))
        [True, True, True, False]
    """
    # Task 2.3
```

We are now able to print the full semantics of a formula: its truth value for every possible model over its variable names. There is a standard way to print this information, called a **truth table**: a table with a line for each possible model, where this line lists each of the truth values of the variable names in the model, and then the truth value of the formula in the model.

Task 4. Implement the missing code for the function `print_truth_table(formula)`, which prints the truth table of the given formula (according to the format demonstrated in the *docstring* of this function).

propositions/semantics.py

```
def print_truth_table(formula: Formula) -> None:
    """Prints the truth table of the given formula, with variable-name columns
    sorted alphabetically.

    Parameters:
        formula: formula to print the truth table of.

    Examples:
        >>> print_truth_table(Formula.parse('~(p&q76)'))
        | p | q76 | ~(p&q76) |
        |---|-----|----------|
        | F | F   | T        |
        | F | T   | T        |
        | T | F   | T        |
        | T | T   | F        |
    """
    # Task 2.4
```

4 Tautologies, Contradictions, and Satisfiability

We are going to pay special attention to two types of formulas—those that get the value *True* in *some* model, and those that get the value *True* in *all* models:

Definition (Satisfiable Formula; Contradiction; Tautology)**.**

- A formula is said to be **satisfiable** if it gets the value *True* in some (at least one) model. A formula that is not satisfiable is said to be a **contradiction**.
- A formula is said to be a **tautology** if it gets the value *True* in all models over its variable names.

For example, the formula '(p&~p)' is a contradiction (why?) and thus is not satisfiable and is certainly not a tautology, while '(p|~p)' is a tautology (why?) and in particular is also satisfiable. The formula '(p&q)' is neither a contradiction nor a tautology, but is satisfiable (why?). Note that a formula ϕ is a contradiction if and only if '~ϕ' is a tautology, and thus a formula ϕ is satisfiable if and only if its negation '~ϕ' is not a tautology. One may figure out whether a given formula satisfies each of these conditions by going over all possible models.

Task 5. Implement the missing code for the three functions `is_tautology(formula)`, `is_contradiction(formula)`, and `is_satisfiable(formula)`, which respectively return whether the given formula is a tautology, is a contradiction, and is satisfiable.

propositions/semantics.py

```
def is_tautology(formula: Formula) -> bool:
    """Checks if the given formula is a tautology.

    Parameters:
        formula: formula to check.

    Returns:
        ``True`` if the given formula is a tautology, ``False`` otherwise.
    """
    # Task 2.5a

def is_contradiction(formula: Formula) -> bool:
    """Checks if the given formula is a contradiction.

    Parameters:
        formula: formula to check.

    Returns:
        ``True`` if the given formula is a contradiction, ``False`` otherwise.
    """
    # Task 2.5b

def is_satisfiable(formula: Formula) -> bool:
    """Checks if the given formula is satisfiable.

    Parameters:
        formula: formula to check.

    Returns:
        ``True`` if the given formula is satisfiable, ``False`` otherwise.
    """
    # Task 2.5c
```

Examples: If `f` is the formula that represents '~(p&q76)', then `is_tautology(f)` should return `False`, `is_contradiction(f)` should return `False`, and `is_satisfiable(f)`

should return `True`. If `g` is the formula that represents '(x|~x)', then `is_tautology(g)` should return `True`, `is_contradiction(g)` should return `False`, and `is_satisfiable(g)` should return `True`.
Hint: Once you implement one of these functions by going over all possible models, it should be easy to use it to implement the other two.

5 Synthesis of Formulas

All of the tasks so far accepted a formula as their input, and answered questions about its truth value in a given model or in some set of models. In the next two tasks you are asked to implement the "reversed" functionality: to take as input desired semantics, and output—synthesize—a formula that conforms to it. Remarkably, this can be done for any desired semantics.

Our first step will be to create a formula whose truth table has a single row with value *True*, with all other rows having value *False*. This can be done in the form of a **conjunctive clause**: a conjunction (i.e., a concatenation using '&' operators) of (one or more) variable names or negation-of-variable-names.

Task 6. Implement the missing code for the function `_synthesize_for_model(model)`, which returns a propositional formula in the form of a conjunctive clause that is *True* for the given model and *False* for every other model over the same variable names.

propositions/semantics.py

```
def _synthesize_for_model(model: Model) -> Formula:
    """Synthesizes a propositional formula in the form of a single conjunctive
    clause that evaluates to ``True`` in the given model, and to ``False`` in
    any other model over the same variable names.

    Parameters:
        model: model over a nonempty set of variable names, in which the
            synthesized formula is to hold.

    Returns:
        The synthesized formula.
    """
    assert is_model(model)
    assert len(model.keys()) > 0
    # Task 2.6
```

Your solution to Task 6 (programmatically) proves the following lemma:

Lemma. *Let $M : S \to \{True, False\}$ be a model over some nonempty finite set of variable names S. There exists a formula in the form of a conjunctive clause that evaluates to True in M and to False in all other models over S.*

If we want to create a formula that has some arbitrary given truth table, that is, that has value *True* for all models in some arbitrary set of models and *False* for any other model, then we can easily just take a disjunction (i.e., a concatenation using '|' operators) of the conjunctive clauses that the above lemma guarantees for each of the models in the set. This would give us a formula in the form called **Disjunctive Normal Form (DNF)**: a disjunction of (one or more) conjunctive clauses.

Task 7 (Programmatic Proof of the DNF Theorem)**.** Implement the missing code for the function `synthesize(variables, values)`, which constructs a propositional formula in DNF from the given description of its truth table.[4]

propositions/semantics.py

```
def synthesize(variables: Sequence[str], values: Iterable[bool]) -> Formula:
    """Synthesizes a propositional formula in DNF over the given variable names,
    that has the specified truth table.

    Parameters:
        variables: nonempty set of variable names for the synthesized formula.
        values: iterable over truth values for the synthesized formula in every
            possible model over the given variable names, in the order returned
            by `all_models(variables)`.

    Returns:
        The synthesized formula.

    Examples:
        >>> formula = synthesize(['p', 'q'], [True, True, True, False])
        >>> for model in all_models(['p', 'q']):
        ...     evaluate(formula, model)
        True
        True
        True
        False
    """
    assert len(variables) > 0
    # Task 2.7
```

Hints: Use the function `_synthesize_for_model` that you implemented in Task 6. Note that the case in which the set of models with value *True* is empty is a special case since we are not allowing to simply return the formula 'F', so you will need to return an equivalent DNF of your choice (over the given variable names).

The fact that you were able to complete Task 7 proves the following rather remarkable theorem:

Theorem (The DNF Theorem)**.** *Let S be a nonempty finite set of variable names. For every Boolean function f over the variable names in S (i.e., f arbitrarily assigns a truth value to every tuple of truth values for the variable names in S) there exists a formula in Disjunctive Normal Form whose truth table is exactly the Boolean function f.*

A Optional Reading: Conjunctive Normal Form

In this section we will consider an alternative approach to synthesizing formulas. While this approach is a precise dual to the one used above, it may be slightly more challenging conceptually. Our first step will be to create a formula whose truth table has all rows except one having value *True*, with the remaining row having value *False*. This can be done in the form of a **disjunctive clause**: a disjunction of (one or more) variable names or negation-of-variable-names.

[4]Once again, readers who implemented Task 3 to return a memory-efficient iterable rather than a `list` are encouraged to implement this method to also accept `values` as an arbitrary iterable.

Optional Task 8. Implement the missing code for the function `_synthesize_for_all_except_model(model)`, which returns a propositional formula in the form of a disjunctive clause that is *False* for the given model and *True* for every other model over the same variable names.

propositions/semantics.py

```
def _synthesize_for_all_except_model(model: Model) -> Formula:
    """Synthesizes a propositional formula in the form of a single disjunctive
    clause that evaluates to ``False`` in the given model, and to ``True`` in
    any other model over the same variable names.

    Parameters:
        model: model over a nonempty set of variable names, in which the
            synthesized formula is to not hold.

    Returns:
        The synthesized formula.
    """
    assert is_model(model)
    assert len(model.keys()) > 0
    # Optional Task 2.8
```

Analogously to the lemma proven by your solution to Task 6, your solution to Optional Task 8 proves the following lemma:

Lemma. *Let $M : S \to \{True, False\}$ be a model over some nonempty finite set of variable names S. There exists a formula in the form of a disjunctive clause that evaluates to False in M and to True in all other models over S.*

If we want to create a formula that has some arbitrary given truth table, that is, that has value *True* for all models in some arbitrary set of models and *False* for any other model, then we can just take a conjunction of the disjunctive clauses that the above lemma guarantees for each of the models *not* (!) in the set. This would give us a formula in the form called **Conjunctive Normal Form (CNF)**: a conjunction of (one or more) disjunctive clauses.

Optional Task 9 (Programmatic Proof of the CNF Theorem)**.** Implement the missing code for the function `synthesize_cnf(variables, values)`, which constructs a propositional formula in CNF from the given description of its truth table.[5]

propositions/semantics.py

```
def synthesize_cnf(variables: Sequence[str], values: Iterable[bool]) -> Formula:
    """Synthesizes a propositional formula in CNF over the given variable names,
    that has the specified truth table.

    Parameters:
        variables: nonempty set of variable names for the synthesized formula.
        values: iterable over truth values for the synthesized formula in every
            possible model over the given variable names, in the order returned
            by `all_models(variables)`.

    Returns:
        The synthesized formula.
```

[5]Once again, readers who implemented Task 3 to return a memory-efficient iterable rather than a `list` are encouraged to implement this method to also accept `values` as an arbitrary iterable.

```
    Examples:
        >>> formula = synthesize_cnf(['p', 'q'], [True, True, True, False])
        >>> for model in all_models(['p', 'q']):
        ...     evaluate(formula, model)
        True
        True
        True
        False
    """
    assert len(variables) > 0
    # Optional Task 2.9
```

Hints: Use the function `_synthesize_for_all_except_model` that you implemented in Optional Task 8. Note that the case in which the set of models with value *False* is empty is a special case since we are not allowing to simply return the formula 'T', so you will need to return an equivalent CNF of your choice (over the given variable names).

The fact that you were able to complete Optional Task 9 proves the following theorem, which is as remarkable as the DNF Theorem:

Theorem (The CNF Theorem)**.** *Let S be a nonempty finite set of variable names. For every Boolean function f over the variable names in S (i.e., f arbitrarily assigns a truth value to every tuple of truth values for the variable names in S) there exists a formula in Conjunctive Normal Form whose truth table is exactly the Boolean function f.*

Attentive readers would notice the **duality** between constructing a DNF formula and a CNF formula. Given a model, `_synthesize_for_model_all_except_model()` returns a disjunctive clause with the exact same structure—only with each variable name replaced by its negation and vice versa (and with *and* operators replaced by *or* operators)—as the conjunctive clause returned by `_syntehsize_for_model()` for the same model. More generally, given a truth table, `_synthesize_cnf()` returns a CNF formula with the exact same structure—only with each variable name replaced by its negation and vice versa (and with *and* operators replaced with *or* operators and vice versa)—as the DNF formula returned by `_syntehsize()` for *the negation of* that truth table. The reason for this is in fact quite simple: these invocations create formulas that are negations of one another, and by De Morgan's laws, the negation of a DNF or CNF formula is a formula with the exact same structure, only with *and* operators replaced with *or* operators and vice versa, and with each variable name replaced by its negation and vice versa.

B Optional Reading: Satisfiability and Search problems

As noted above, in this book we will mostly ignore questions of computational complexity, i.e., of the running time of the algorithms that we use, and usually focus only on correctness, which is what guarantees the validity of the corresponding proofs. However, since handling truth values of propositional formulas turns out to be a central issue of study in computational complexity theory, we will now shortly discuss these computational complexity aspects.

Looking at the various tasks of this chapter, one may observe that there are two very different levels of running times involved. Tasks that deal with a single model, like `evaluate(formula, model)` or `_synthesize_for_model(model)` (or

`_synthesize_for_all_except_model(model)`), can easily be solved very efficiently and your solution should easily work for formulas with thousands or even millions of variable names. On the other hand, as we have noted, tasks that deal with *all* models of a given formula, like `print_truth_table(formula)` or `synthesize(models, values)` (or `synthesize_cnf(models, values)`), have inputs or outputs whose size is exponential in the number of variable names used in the formula. Thus, it is certain that there is no hope that *any* solution of any of these latter tasks could operate on formulas that have more than a few dozen variable names, as otherwise the input and/or output would not be able to fit into your computer's memory.

The most interesting tasks to study from the point of view of their computational complexity, though, are rather tasks such as implementing `is_tautology(formula)` or `is_satisfiable(formula)`, whose inputs and outputs don't explicitly involve the set of all models and yet the natural algorithm for solving them is precisely to go over all possible models (as your solution does). In these cases the natural algorithm that goes over all models will take an exponential amount of time and thus this algorithm would not be able to handle formulas with more than a few dozen variable names. Are there other algorithms for determining whether a given formula is satisfiable (or equivalently, whether a given formula is tautology[6]) that are more efficient than simply trying each and every possible model? It is true that for some special classes of formulas, there exists such an algorithm. For example, efficiently checking whether a DNF formula is satisfiable turns out to be possible and even quite easy[7] (and dually, so does checking whether a CNF formula is a tautology[8]). Furthermore, much effort was spent on the problem of efficiently finding a satisfying assignment to a formula, mostly using clever backtracking techniques, and there are programs that, *heuristically*, work rather well on formulas encountered when translating some real-world challenges into satisfiability of propositional formulas. But does there exist an algorithm that is *guaranteed* to efficiently find a satisfying solution if such exists? While this may seem to be a rather specific algorithmic question that may interest few people who are not logicians, it turns out that quite the opposite is true. In fact, the far-reaching implications of this problem cannot be overstated, since an immense variety of computational problems can be converted into a question of whether some propositional formula is satisfiable or not.

A paradigmatic example of such a "conversion" is that of **computational search problems**. A typical such computational problem asks to find some kind of entity x that satisfies a certain set of given properties. To solve such a task, an algorithm may encode the required x as a sequence of bits $x_1, \ldots, x_n$ and encode the required properties as a propositional formula over these variable names such that a satisfying assignment encodes the desired solution. To make this concrete, let's take as an example the **Graph Coloring Problem**: given a graph, find a **graph coloring** of the vertices of the graph, using at most k colors, such that no two vertices connected by an edge are colored by the same.

[6]We write "equivalently" since a formula is satisfiable if and only if its negation is not a tautology. This immediately tells us that the existence of an efficient algorithm for checking whether a formula is a tautology would be equivalent (and as hard to find, if one exists) to the existence of an efficient algorithm for satisfiability, since an algorithm for any one of these problems could easily be converted to an algorithm for the other problem by checking the negation of the input formula.

[7]Indeed, it is not hard to verify that a DNF formula is NOT satisfiable if and only if every one of its clauses is NOT satisfiable, which in turn occurs if and only if every one of its clauses contains both a variable name and its own negation.

[8]Indeed, dually to checking the satisfiability of a DNF formula, it is not hard to verify that a CNF formula is a tautology if and only if every one of its clauses is a tautology, which in turn occurs if and only if every one of its clauses contains both a variable name and its own negation.

The file `propositions/reductions.py`, which contains all of the functions that you are asked to implement in this section, already contains the definition of how we will represent graphs in Python in this section:

propositions/reductions.py

```
#: A graph on a vertex set of the form (1,...,`n_vertices`), represented by the
#: number of vertices `n_vertices` and a set of edges over the vertices.
Graph = Tuple[int, AbstractSet[Tuple[int, int]]]

def is_graph(graph: Graph) -> bool:
    """Checks if the given data structure is a valid representation of a graph.

    Parameters:
        graph: data structure to check.

    Returns:
        ``True`` if the given data structure is a valid representation of a
        graph, ``False`` otherwise.
    """
    (n_vertices, edges) = graph
    for edge in edges:
        for vertex in edge:
            if not 1 <= vertex <= n_vertices:
                return False
        if edge[0] == edge[1]:
            return False
    return True
```

This file also contains the function `is_valid_3coloring(graph, coloring)`, which we have already implemented for you and which verifies that the given coloring is a valid coloring of the given graph:

propositions/reductions.py

```
def is_valid_3coloring(graph: Graph, coloring: Mapping[int, int]) -> bool:
    """Checks whether the given coloring is a valid coloring of the given graph
    by the colors 1, 2, and 3.

    Parameters:
        graph: graph to check.
        coloring: mapping from the vertices of the given graph to colors, to
            check.

    Returns:
        ``True`` if the given coloring is a valid coloring of the given graph by
        the colors 1, 2, and 3; ``False`` otherwise.
    """
    assert is_graph(graph)
    (n_vertices, edges) = graph
    for vertex in range(1, n_vertices + 1):
        if vertex not in coloring.keys() or coloring[vertex] not in {1, 2, 3}:
            return False
    for edge in edges:
        if coloring[edge[0]] == coloring[edge[1]]:
            return False
    return True
```

A coloring of the graph can be encoded by having, for every vertex v and possible

color c, a Boolean variable x_{vc} that represents that vertex v is colored by color c, and then the constraints that state that an assignment of values to the variables names—a model—represents a valid coloring of the graph can be expressed as whether a certain propositional formula over these variable names is satisfied by this model.

Optional Task 10. Implement the missing code for the two functions `graph3coloring_to_formula(graph)` and `assignment_to_3coloring(graph, assignment)`. The former function returns a propositional formula that "encodes" the 3-coloring problem of the given graph in the sense that it is satisfiable if and only if that graph is colorable by at most three colors. Moreover, we require that any satisfying assignment to the generated formula can actually be converted into a 3-coloring of the graph, which the latter function accomplishes. Your implementation should be computationally efficient in the sense that each of these functions should be able to easily handle graphs with hundreds or even thousands of vertices.

propositions/reductions.py

```
def graph3coloring_to_formula(graph: Graph) -> Formula:
    """Efficiently reduces the 3-coloring problem of the given graph into a
    satisfiability problem.

    Parameters:
        graph: graph whose 3-coloring problem to reduce.

    Returns:
        A propositional formula that is satisfiable if and only if the given
        graph is 3-colorable.
    """
    assert is_graph(graph)
    # Optional Task 2.10a

def assignment_to_3coloring(graph: Graph, assignment: Model) -> \
        Mapping[int, int]:
    """Efficiently transforms an assignment to the formula corresponding to the
    3-coloring problem of the given graph, to a 3-coloring of the given graph so
    that the 3-coloring is valid if and only if the given assignment is
    satisfying.

    Parameters:
        graph: graph to produce a 3-coloring for.
        assignment: assignment to the variable names of the formula returned by
            `graph3coloring_to_formula(graph)`.

    Returns:
        A 3-coloring of the given graph by the colors 1, 2, and 3 that is valid
        if and only if the given assignment satisfies the formula
        `graph3coloring_to_formula(graph)`.
    """
    assert is_graph(graph)
    formula = graph3coloring_to_formula(graph)
    assert evaluate(formula, assignment)
    # Optional Task 2.10b
```

Once the above two functions are implemented, we get an algorithm for 3-coloring a graph:

propositions/reductions.py

```
def tricolor_graph(graph: Graph) -> Union[Mapping[int, int], None]:
    """Computes a 3-coloring of the given graph.

    Parameters:
        graph: graph to 3-color.

    Returns:
        An arbitrary 3-coloring of the given graph if it is 3-colorable,
        ``None`` otherwise.
    """
    assert is_graph(graph)
    formula = graph3coloring_to_formula(graph)
    for assignment in all_models(list(formula.variables())):
        if evaluate(formula, assignment):
            return assignment_to_3coloring(graph, assignment)
    return None
```

Your solution to Optional Task 10 was required to be computationally efficient;[9] it is probably able to easily handle graphs with thousands of vertices and formulas with thousands of variable names. The only efficiency bottleneck in the above algorithm for coloring a graph is in the main loop of the function `tricolor_graph(graph)`:

```
for assignment in all_models(list(formula.variables())): ...
```

There are exponentially many possible models and this directly means that once the number of variable names is more than a few dozen, this algorithm becomes impractical. But, an efficient algorithm for finding a satisfying assignment (if such exists) to a formula could immediately be used to replace this loop and yield an efficient algorithm for graph coloring. As it turns out, not only can the coloring problem be efficiently converted—**reduced**—to the formula satisfiability problem, but so can also an immense gamut of other diverse and seemingly unrelated problems. In fact, Cook's Theorem, a central result in the field of **Computational Complexity**, states that *any* computational search problem can be efficiently reduced to finding a satisfying assignment to a propositional formula. This class of problems includes most computational tasks of interest including, e.g., the traveling salesmen problem, breaking any cryptographic code, solving arbitrary sets of equations, automatically proving theorems, finding bugs in programs, and much more. An enormous amount of effort has gone into trying to find efficient algorithms for many of these problems, all in vain. Since an efficient algorithm for satisfiability would, in one fell swoop, provide efficient algorithms for all of these, the existence of one seems rather unlikely.

In the notation of computational complexity, the class of efficiently solvable problems (formally, those that have a polynomial-time algorithm) is called P, and the class of computational search problems is called NP. The fact that any problem in NP can be reduced to a satisfiability problem is stated as satisfiability being ***NP*-complete**. So, an efficient algorithm for determining whether a given formula is satisfiable exists if and only if *every* problem in NP also has an efficient algorithm, i.e., if and only if "$P = NP$". The question of whether "$P = NP$?" is arguably considered to be the main open problem in Computer Science, and as mentioned above the general belief is that the answer is negative, i.e., that satisfiability does not have an efficient algorithm.

Recall that we said earlier that efficiently checking whether a DNF formula is satisfiable (or dually, whether a CNF formula is a tautology) is in fact easy. Supposedly, one could

[9] Our test for this task takes a while to run, though, since the test—not the task—enumerates over all possible models to check the solution.

have then proposed the following algorithm for satisfiability of an arbitrary formula: first convert the formula to an equivalent formula (i.e., with the same truth table) in DNF, and then efficiently check whether that formula is satisfiable. The problem here is that we know of no efficient way of converting a formula into DNF (or into CNF). We do know, however, given an arbitrary formula ϕ, to efficiently construct a formula ϕ' in CNF such that while these two formulas may not be equivalent, it does hold that ϕ' is satisfiable if and only if ϕ is, and furthermore, given a satisfying assignment for ϕ', it is easy to efficiently find a satisfying assignment for ϕ. This of course immediately implies that satisfiability continues to be as difficult even when we restrict the input formula to be in CNF.[10]

In fact, for a similar reason, satisfiability continues to be as difficult even when restricting to **3-CNF**: the special case of CNF where each clause is a disjunction of *at most three* variables names or negation-of-variable-names, e.g., '(((((p|q)|~r)&(~p|~s))&((q|s)|~r))&t)'.[11] The significance of this fact is that it gives us a general way for proving that other computational search problems are *NP*-complete: 3-CNF formulas intuitively are easier to reason about than general formulas, and this helped computer scientists to show over the years that the question of whether a 3-CNF formula is satisfiable can be reduced into a variety of computational search problems (i.e., to show that an efficient algorithm for any of a variety of computation search problems could be used to construct an efficient algorithm for satisfiability). Exhibiting such a reduction for some computational search problem Q implies that the problem Q itself is *NP*-complete, and thus cannot be solved efficiently unless $P = NP$. Thousands of problems have been proven to be *NP*-complete this way, including many optimization problems, for example the traveling salesman problem, the knapsack problem, and many other problems including even the 3-coloring problem discussed above.[12] So, finding an efficient algorithm for the seemingly innocent task of 3-coloring a graph would immediately give an efficient algorithm for any NP-complete problem, including breaking any cryptographic code!

DRAFT

[10]Recall however that for formulas in CNF it is easy to tell whether they are tautologies: the only way that a CNF formula may be a tautology is is if each of its clauses contains some of variable name and its own negation. The special form that retains the hardness of the tautology problem is the dual DNF. This duality is an implication (why?) of the above-discussed connection between the problems of satisfiability and checking whether a given formula is a tautology.

[11]And dually, of course, for the above reasons (why?), tautology continues to be as difficult even when restricting to 3-DNF: the special case of DNF where each clause is a conjunction of at most three variable names or negation-of-variable-names.

[12]We emphasize that this reduction is in the opposite direction to the reduction shown in Optional Task 10 above, which is from the 3-coloring problem to satisfiability: there we used an algorithm for satisfiability to construct an algorithm for 3-coloring a graph.