Chapter 2:

Propositional Logic Semantics

In the previous chapter we were careful not to assign any meaning, any **semantics**, to propositional formulae. In contrast, this chapter does exactly that. Our intention for these semantics is the following: every variable (e.g. 'p' or 'q76') will stand for a certain **atomic proposition** that may be either true or false, independently of the other atomic propositions. A formula will describe a more complex proposition, whose truth or lack thereof depends on which combination of atomic propositions is true. For example, we may have 'p' represent "It is raining," 'q' represent "I have an umbrella," and 'r' represent "I am wet." A Formula like '((p&~q)→r)' states that if it is raining and I do not have an umbrella then it must be that I am wet. The point of such a formalism is that we will later be able to deduce, from this formula, related further propositions such as '((~r&p)→q)' (If I am not wet and it is raining then it must be that I have an umbrella), in a purely syntactic logical way.

1   Models and Truth Values

Formally, the semantics we will give to a formula is the truth value that it gets in every possible setting of its atomic propositions. We view a possible setting of these atomic propositions 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 atomic propositions (variables). A **model** M over S is a function that assigns a truth value to every variable in S. That is, $M : S \to \{\mathit{True}, \mathit{False}\}$.

The file `propositions/semantics.py` deals with the semantics of propositional formulae. 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 to a Boolean value:

```
━━━━━━━━━ propositions/semantics.py ━━━━━━━━━
#: A mapping from variable names to truth values.
Model = Mapping[str, bool]

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

    Parameters:
        model: dictionary to check.

    Returns:
        ``True`` if the given dictionary is a model over some set of variables,
        ``False`` otherwise.
    """
```

```
        for key in model:
            if not (is_variable(key) and type(model[key]) is bool):
                return False
        return True

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

    Parameters:
        model: model to check.

    Returns:
        A set of all variables 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 variables that contains all those used in ϕ, we define the **(truth) value** of the formula ϕ in the model M inductively 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 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 value of ψ is *False* (and is *False* otherwise).

- If ϕ is of the form '$(\psi\&\xi)$', then its value is *True* if the value of both ψ and ξ is *True* (and is *False* otherwise); if ϕ is of the form '$(\psi|\xi)$', then its value is *True* if the value of either ψ or ξ is *True* (and is *False* otherwise); if ϕ is of the form '$(\psi{\rightarrow}\xi)$', then its value is *True* if either the value of ψ is *False* or if the value of ξ 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* ("I do NOT have an umbrella"), and $M('r') = $ *True* ("I am wet"), and in this model the formula '$((p\&\sim q){\rightarrow}r)$' evaluates to the value *True* as defined recursively: '$\sim q$' evaluates to *True* (since $M('q') = $ *False* in the model), and so '$(p\&\sim q)$' evaluates to *True* (since both 'p' and '$\sim q$' evaluate to *True*) and finally '$((p\&\sim q){\rightarrow}r)$' evaluates to the value *True* (since both sides of the $\rightarrow$ operator 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*.

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 variables of the 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
```

2   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 variables, then there are exactly
2^n possible models over this set: all possible combinations where each of the variables is
set 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 wrote so far would
have no problem dealing with formulae that handle millions of variables, once we want
to list *all* the possible models of a given set of variables, we will not be able to handle
more than a few dozen variables at most: already with 40 variables 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[1] of all possible models over the given variables.

```
————————————————— propositions/semantics.py —————————————————
def all_models(variables: List[str]) -> Iterable[Model]:
    """Calculates all possible models over the given variables.

    Parameters:
        variables: list of variables over which to calculate the models.

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

    Examples:
        >>> list(all_models(['p', 'q']))
```

---

[1] 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. While 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 accepts any iterable, and not merely a `list`. We do not, however, intend to run any of
this code on more than a few variables, so we do not make any efficiency requirements from your code,
and we do allow solving this task by returning a `list` of all models.

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

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.[2]

<div align="center">propositions/semantics.py</div>

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

    Parameters:
        formula: formula to calculate the truth value of.
        model: 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 of its variables. 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 atomic propositions 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).

<div align="center">propositions/semantics.py</div>

```
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)'))
```

---

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

```
      | p | q76 | ~(p&q76) |
      |---|-----|----------|
      | F | F   | T        |
      | F | T   | T        |
      | T | F   | T        |
      | T | T   | F        |
    """
    # Task 2.4
```

3   Tautologies, Contradictions, and Satisfiability

We are going to pay special attention to two types of formulae: 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.

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 $\sim\phi$ is a tautology, and thus a formula ϕ is satisfiable if and only if $\sim\phi$ 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`; while 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.

4 Synthesis of Formulae

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. The next two tasks implement the "reversed" task: they take as input desired semantics, and output — synthesize — a formula that conforms to it.

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 **clause**: a conjunction (i.e., a concatenation using '&' operators) of (one or more) variables or negation-of-variables.

Task 6. Implement the missing code for the function `synthesize_for_model(model)`, which returns a propositional formula in the form of a clause that has a true value for the given model and a false value for every other model over the same variables.

```
                        propositions/semantics.py
def synthesize_for_model(model: Model) -> Formula:
    """Synthesizes a propositional formula in the form of a single clause that
        evaluates to ``True`` in the given model, and to ``False`` in any other
        model over the same variables.

    Parameters:
        model: model in which the synthesized formula is to hold.

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

Your solution to Task 6 proves the following lemma:

Lemma. *Let $M : S \to \{True, False\}$ be a model over some finite set of variables S. Then there exists a formula in the form of a 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 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 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) clauses.

Task 7. 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.[3]

```
                        _____ propositions/semantics.py _____
def synthesize(variables: List[str], values: Iterable[bool]) -> Formula:
    """Synthesizes a propositional formula in DNF over the given variables,
    that has the specified truth table.

    Parameters:
        variables: nonempty set of variables for the synthesized formula.
        values: iterable over truth values for the synthesized formula in every
            possible model over the given variables, 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 where 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 variables).

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

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

---

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

5   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, as it turns out that handling truth values of Boolean formulae is a central issue of study in computational complexity theory, we will now shortly discuss these computational complexity aspects.

Looking at the different 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)`, can easily be solved very efficiently and your solution should easily work for formulae with thousands or even millions of variables. 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)`, have inputs or outputs whose size is exponential in the number of variables used in the formula. Thus, it is certain that there is no hope of being able to call any of these latter tasks to operate on formulae that have more than a few dozen variables, 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 formulae with more than a few dozen variables. Are there other algorithms for determining whether a given formula is satisfiable (or equivalently, whether a given formula is tautology[4]) that are more efficient than simply trying each and every possible model? It is true that for some special classes of formulae, there exists such an algorithm (for example, efficiently checking whether a DNF formula is satisfiable turns out to be possible and even quite easy[5]). 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 formulae encountered when translating some real-world challenges into satisfiability of propositional formulae. 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 Boolean 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

---

[4]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.

[5]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 contain both a variable and its own negation.

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 variables 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 **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. A coloring of the graph can be encoded by having, for every vertex v and possible color c, a Boolean variable—atomic proposition—x_{vc} that represents that vertex v is colored by color c, and then the constraints that state that a certain assignment of values to the variables—a certain model—represents a valid coloring of the graph can be expressed as a propositional formula over these atomic propositions.

Optional Task 8. Implement the missing code for the two functions `graph3coloring_to_formula(graph)` and `assignment_to_3coloring(graph, assignment)` (in the file `propositions.reductions.py`). 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. We requires that your implementation is 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 ───────────────
#: 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, Set[Tuple[int, int]]]

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.
    """
    (n_vertices, edges) = graph
    for edge in edges:
        assert len(edge) == 2
        for vertex in edge:
            assert 1 <= vertex <= n_vertices
    # Optional Task 2.8a

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, into 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: an assignment to the variables 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)`.
    """
    (n_vertices, edges) = graph
    formula = graph3coloring_to_formula(graph)
    assert evaluate(formula, assignment)
    # Optional Task 2.8b
```

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.
    """
    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 8 was required to be computationally efficient; it is probably able to easily handle graphs with thousands of vertices and formulae with thousands of variables. 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 variables is more than a few dozen, then this algorithm becomes impractical. But, an efficient algorithm for finding a satisfying assignment 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 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 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, this 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 is in fact easy. Supposedly, one could 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. We do know, however, given an arbitrary formula ϕ, to efficiently construct a formulae ϕ' in a special form called CNF (**Conjunctive Normal Form**: a conjunction of clauses, each of which is a disjunction of variables or negation-of-variables), such that while these two formulae 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 assumption 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.[6]

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 or negation-of-variables, e.g., '$(((((p|q)|\sim r)\&(\sim p|\sim s))\&((q|s)|\sim r))\&t)$'.[7] 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 formulae intuitively are easier to reason about than general formulae, 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 a 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 proved 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.[8] So, finding an efficient algorithm for the seemingly innocent task of

---

[6]Note however that for formulae in CNF form 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 a variable 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.

[7]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 variables or negation-of-variables.

[8]We emphasize that this reduction is in the opposite direction to the reduction shown in Optional Task 8 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.

3-coloring a graph would immediately give an efficient algorithm for any NP-complete problem, including breaking any cryptographic code!