Chapter 11:

The Deduction Theorem and Prenex Normal Form

In this chapter, we will prove two important and general tools, which beyond being useful each on its own, will also be of utmost importance in our proof of the Completeness Theorem for Predicate Logic in the next chapter, in which our analysis of this entire book will culminate:

- The **Deduction Theorem** (for Predicate Logic), which shows (much like the Deduction Theorem for Propositional Logic, which you proved in Chapter 5) that if some statement ξ is provable (in Predicate Logic) from some assumptions that include an assumption ψ, then under certain mild conditions, the statement '$(\psi\rightarrow\xi)$' is provable from the same set of assumptions, without ψ. From the Deduction Theorem, much like in Propositional Logic, we will derive a theorem on **Soundness of Proofs by Way of Contradiction** in Predicate Logic.

- The **Prenex Normal Form Theorem**, which shows that every formula can be transformed into an equivalent formula in **prenex normal form**, that is, a formula where all quantifiers appear at the beginning (top levels) of the formula.

1 The Deduction Theorem

Recall that in Chapter 5, you have proved the Deduction Theorem for Propositional Logic, and in particular its "hard direction" (which is often referred to by itself as the Deduction Theorem for Propositional Logic):

Theorem ("Hard Direction" of The Deduction Theorem for Propositional Logic)**.** *Let $\mathcal{R}$ be a set of inference rules that includes MP, I1, and D, and may additionally include only inference rules with no assumptions. Let A be an arbitrary set of formulas, and let ϕ and ψ be two additional formulas. If $A\cup\{\phi\}\vdash_{\mathcal{R}}\psi$, then $A\vdash_{\mathcal{R}}$ '$(\phi\rightarrow\psi)$'.*

This establishes that under certain conditions, the converse of MP holds as well: not only can we prove ψ from some assumptions that include ϕ if we can prove '$(\phi\rightarrow\psi)$' from the same assumptions without ϕ, but also conversely, if we can prove ψ from some assumptions that include ϕ, then (under mild conditions on the inference rules that may be used), we can prove '$(\phi\rightarrow\psi)$' from the same assumptions without ϕ.

Since this theorem turned out to be an extremely useful tool in writing proofs in Propositional Logic, and in particular in proving the Tautology Theorem and therefore the Completeness Theorem for Propositional Logic, we will now prove an analogue of this theorem for Predicate Logic. Since our proof system in Predicate Logic does not allow for any inference rules that have assumptions except MP and UG, we will not need any analogue of the condition from the above theorem regarding which inference rules may have assumptions. The condition from that theorem on the set of inference rules containing MP, I1, and D will be replaced, strictly speaking, by being able to use MP, UG, and all tautologies (which all of our proofs can. . .), and the axiom US, which was introduced in the previous chapter (but which we have in fact so far not used at all!).

We will also have an additional new condition that has no analogue in Propositional Logic: that the proof of ψ from the set of assumptions that includes ϕ does not involve UG over any variable name that is free in ϕ. Unlike the former conditions, which are technical, this new condition is crucial. To see this, notice for example that from only the assumption 'y=0' (which remember that we semantically interpret as "for any assignment to 'y' of an element from the universe of the model, it holds that that element is the interpretation of '0'") one can use UG over the variable name 'y' to deduce '∀y[y=0]' (which has the exact same semantic interpretation). Nonetheless, it is not the case that without any assumptions whatsoever, one can prove '(y=0→∀y[y=0])' (which remember that semantically we interpret as "if any element is the interpretation of zero, then this implies that all elements are the interpretation of zero"), but this would have been implied by the Deduction Theorem that we will state below if it were not for the constraint on UG. Indeed, in all but very specific models (those with only one element) the formula '(y=0→∀y[y=0])' is simply not satisfied. More generally, if one can prove from 'R(x)' that 'Q(x)', then due to our convention on interpreting assumptions/conclusions with free variable names, this proof in fact shows that '(∀x[R(x)]→∀x[Q(x)])' rather than that (for every 'x' separately) '(R(x)→Q(x))', which has a very different semantic interpretation. Our main uses of the Deduction Theorem for Predicate Logic will however be when ϕ is a **sentence**—a formula with no free variable names at all—in which case this constraint is always trivially satisfied regardless of the structure of the proof of ψ from (among other assumptions) ϕ, so we will not have to worry about this constraint in that case. We will now formally state this theorem (or rather, only its "hard part," since its "easy part" follows from the ability to use MP precisely as in Propositional Logic):

Theorem (The Deduction Theorem for Predicate Logic). *Let A be a set of schemas that includes US, and let ϕ and ψ be two formulas. If $A \cup \{\phi\} \vdash \psi$ via a proof that does not use UG over any variable name that is free in ϕ, then $A \vdash$ '$(\phi \to \psi)$'.*

You will now prove this theorem. All of the functions that you are asked to implement in this section are contained in the file `predicates/deduction.py`.

Task 1 (Programmatic Proof of the Deduction Theorem). Implement the missing code for the function `remove_assumption(proof, assumption)`. This function takes as input a (predicate-logic) proof of some conclusion ψ that:

- has our six logical axiom schemas[1] among its assumptions/axioms,
- has the given assumption ϕ as a simple assumption/axiom (with no templates), and
- does not use UG over any variable name that is free in ϕ.

The function returns a proof of '$(\phi \to \psi)$' from the same assumptions/axioms except ϕ.

[1]While we strictly only need US, we assume all six logical axiom schemas so that you may easily use the `Prover` class in your solution.

predicates/deduction.py

```
def remove_assumption(proof: Proof, assumption: Formula,
                      print_as_proof_forms: bool = False) -> Proof:
    """Converts the given proof of some `conclusion` formula, an assumption of
    which is `assumption`, to a proof of '(`assumption`->`conclusion`)' from the
    same assumptions except `assumption`.

    Parameters:
        proof: valid proof to convert, from assumptions/axioms that include
            `Prover.AXIOMS`.
        assumption: formula that is a simple assumption (i.e., without any
            templates) of the given proof, such that no line of the given proof
            is a UG line over a variable name that is free in this assumption.
        print_as_proof_forms: flag specifying whether the proof of
            '(`assumption`->`conclusion`)' is to be printed in real time as it
            is being created.

    Returns:
        A valid proof of '(`assumption`->`conclusion`)' from the same
        assumptions/axioms as the given proof except `assumption`.
    """
    assert proof.is_valid()
    assert Schema(assumption) in proof.assumptions
    assert proof.assumptions.issuperset(Prover.AXIOMS)
    for line in proof.lines:
        if isinstance(line, Proof.UGLine):
            assert line.formula.variable not in assumption.free_variables()
    # Task 11.1
```

Hints: As in Chapter 5, for each formula ξ that is deduced by some line of the original proof, the new proof should deduce '$(\phi\rightarrow\xi)$'; that is, the suggested strategy is to create your proof by going over the lines of the given proof, in order, and whenever a line of the given proof deduces some ξ, to add one or more lines to the new proof that together deduce '$(\phi\rightarrow\xi)$'. Use tautological implications generously; when the original proof line uses UG, you will need to work a bit harder—try to understand how to use US in this case.

As we have seen in Chapter 5, one important implication of (any version of) the Deduction Theorem is that we can use it to formally explain why a **proof by way of contradiction** works. Recall that in a proof by way of contradiction, we make an assumption, use it to prove the negation of an axiom (recall that we encourage you to think of a proof by way of contradiction as a **proof by way of contradicting an axiom**), and deduce from this that the assumption that we made is in fact false. Since in Predicate Logic (thanks to your proof of the Tautology Theorem in Propositional Logic) we can use all tautologies as axioms of our proofs, we will allow the "proof of the negation of an axiom" that we are given to be a proof of any **contradiction**,[2] i.e., of the negation of any tautology.[3]

Theorem (Soundness of Proofs by Way of Contradiction in Predicate Logic)**.** *Let A be a set of schemas that includes US, and let ϕ be a formula. If a contradiction (i.e., negation*

[2]In some sense, this may be thought of as further justifying the common name "proof by (way of) contradiction."

[3]Similarly to Chapter 5, it is possible also in predicate logic to have other equivalent definitions for this notion of inconsistency, and in particular ones that do not involve tautologies. For example, since every formula is a tautological implication of a contradiction, the ability to prove a contradiction from certain assumptions/axioms is equivalent to the ability to prove every formula from the same assumptions/axioms, so this notion of inconsistency can be phrased in a purely syntactic way and is precisely the same as in propositional logic.

of a tautology) is provable from $A \cup \{\phi\}$ without using UG over any variable name that is free in ϕ, then $A \vdash$ '$\sim\phi$'.

You will now prove this theorem. The proof will once again be considerably simpler than the proof for Propositional Logic due to the ability to use all tautologies freely at this point.

Task 2 (Programmatic Proof of the theorem on Soundness of Proofs by Way of Contradiction)**.** Implement the missing code for the function `prove_by_way_of_contradiction( proof, assumption)`. This function takes as input a (predicate-logic) proof of some contradiction that:

- has our six logical axiom schemas[4] among its assumptions/axioms,
- has the given assumption ϕ as a simple assumption/axiom (with no templates), and
- does not use UG over any variable name that is free in ϕ.

The function returns a proof of '$\sim\phi$' from the same assumptions/axioms except ϕ.

predicates/deduction.py

```
def prove_by_way_of_contradiction(proof: Proof, assumption: Formula) -> Proof:
    """Converts the given proof of a contradiction, an assumption of which is
    `assumption`, to a proof of '~`assumption`' from the same assumptions except
    `assumption`.

    Parameters:
        proof: valid proof of a contradiction (i.e., a formula whose negation is
            a tautology) to convert, from assumptions/axioms that include
            `Prover.AXIOMS`.
        assumption: formula that is a simple assumption (i.e., without any
            templates) of the given proof, such that no line of the given proof
            is a UG line over a variable name that is free in this assumption.

    Returns:
        A valid proof of '~`assumption`' from the same assumptions/axioms as the
        given proof except `assumption`.
    """
    assert proof.is_valid()
    assert Schema(assumption) in proof.assumptions
    assert proof.assumptions.issuperset(Prover.AXIOMS)
    for line in proof.lines:
        if isinstance(line, Proof.UGLine):
            assert line.formula.variable not in assumption.free_variables()
    # Task 11.2
```

Guidelines: Recall that `Proof` objects are immutable. Therefore, you cannot simply add lines to any `Proof` object. Instead, construct a new `Prover` object, use the `add_proof()` method of that object (which we have already implemented for you) to insert all the lines of any `Proof` object into the proof of this prover, and then add more lines to that proof using the `Prover` API, including the methods that you implemented in Chapter 10 (and in particular, `add_tautological_implication()`).

[4]Once again, we strictly only need US.

2 Prenex Normal Form

According to Wiktionary (“prenex,” 2021),[5] the term “prenex” comes from the Late Latin *praenexus*, meaning “bound up in front.” A formula is said to be in **prenex normal form** if all of its quantifications are at the beginning / at the top-most levels. More precisely:

Definition (Prenex Normal Form)**.** A formula is said to be in **prenex normal form** if there is an initial sequence of nodes of the formula that all contain quantifications, and all other nodes in the formula do not contain quantifications.

For example, the formula ‘∀x[∃y[(R(x,y)→Q(y))]]’ is in prenex normal form, while the formula ‘∀x[(R(x)→∃y[Q(x,y)])]’ is *not* in prenex normal form since the existential quantification is only on the second part of the implication (or more precisely, the *implies* node, which is not a quantification node, precedes the node of that quantification). The formula ‘∀x[(∃y[R(x,y)]→Q(x))]’ is similarly *not* in prenex normal form since the existential quantification is only on the first part of the implication (and again the *implies* node precedes it). In this section, we will explore this form. The functions that you are asked to implement in this section are contained in the file `predicates/prenex.py` unless otherwise noted.

Task 3. Implement the missing code for the function `is_in_prenex_normal_form(formula)`, which checks whether a given formula is in prenex normal form.

predicates/prenex.py
```
def is_quantifier_free(formula: Formula) -> bool:
    """Checks if the given formula contains any quantifiers.

    Parameters:
        formula: formula to check.

    Returns:
        ``False`` if the given formula contains any quantifiers, ``True``
        otherwise.
    """
    # Task 11.3a

def is_in_prenex_normal_form(formula: Formula) -> bool:
    """Checks if the given formula is in prenex normal form.

    Parameters:
        formula: formula to check.

    Returns:
        ``True`` if the given formula in prenex normal form, ``False``
        otherwise.
    """
    # Task 11.3b
```

Guidelines: First implement the missing code for the recursive function `is_quantifier_free(formula)`, which checks whether a given formula contains no quantifiers, and then use that function to solve this task.

[5]Wiktionary, *The Free Dictionary*, s.v. “prenex,” (accessed June 22, 2021), `https://en.wiktionary.org/w/index.php?title=prenex&oldid=60445646`.

As you will show in this section, and this will be incredibly useful in the next chapter, every formula can be converted into prenex normal form, and moreover, the equivalence of the original formula and the one in prenex normal form is provable by a predicate-logic proof via just our four logical axiom schemas of quantification from Chapter 10:

Theorem (The Prenex Normal Form Theorem)**.** *For every formula ϕ there exists an equivalent formula ψ in prenex normal form, such that the equivalence between ϕ and ψ (i.e., '$((\phi\rightarrow\psi)\&(\psi\rightarrow\phi))$') is provable from UI, EI, US, and ES.*

The idea of this conversion is to use various logical equivalences to "pull out" the quantifications. This becomes doable if we make sure that quantified variable names in different parts of the formula do not "clash" with one another, nor with the free variable names of the formula, so that each variable name occurrence *remains* bound by the same quantifier, or *remains* free, before and after "pulling out" the quantifications, despite the scopes of the various quantifications having changed. Specifically, let x be a variable name, let $\phi(x)$ be an arbitrary formula, and let ψ be a formula that *does not have x as a free variable name.* Then the following logical equivalences can be used to "pull out" the quantifications:

1. '$\sim\forall x[\phi(x)]$' is equivalent to '$\exists x[\sim\phi(x)]$'.
2. '$\sim\exists x[\phi(x)]$' is equivalent to '$\forall x[\sim\phi(x)]$'.
3. '$(\forall x[\phi(x)]\&\psi)$' is equivalent to '$\forall x[(\phi(x)\&\psi)]$'.
4. '$(\exists x[\phi(x)]\&\psi)$' is equivalent to '$\exists x[(\phi(x)\&\psi)]$'.
5. '$(\psi\&\forall x[\phi(x)])$' is equivalent to '$\forall x[(\psi\&\phi(x))]$'.
6. '$(\psi\&\exists x[\phi(x)])$' is equivalent to '$\exists x[(\psi\&\phi(x))]$'.
7. '$(\forall x[\phi(x)]|\psi)$' is equivalent to '$\forall x[(\phi(x)|\psi)]$'.
8. '$(\exists x[\phi(x)]|\psi)$' is equivalent to '$\exists x[(\phi(x)|\psi)]$'.
9. '$(\psi|\forall x[\phi(x)])$' is equivalent to '$\forall x[(\psi|\phi(x))]$'.
10. '$(\psi|\exists x[\phi(x)])$' is equivalent to '$\exists x[(\psi|\phi(x))]$'.
11. '$(\forall x[\phi(x)]\rightarrow\psi)$' is equivalent to '$\exists x[(\phi(x)\rightarrow\psi)]$'.
12. '$(\exists x[\phi(x)]\rightarrow\psi)$' is equivalent to '$\forall x[(\phi(x)\rightarrow\psi)]$'.
13. '$(\psi\rightarrow\forall x[\phi(x)])$' is equivalent to '$\forall x[(\psi\rightarrow\phi(x))]$'.
14. '$(\psi\rightarrow\exists x[\phi(x)])$' is equivalent to '$\exists x[(\psi\rightarrow\phi(x))]$'.

Each of these equivalences "pulls" a quantification "out" of a logical operator, so multiple applications of these equivalences will pull all of the quantifications outside. When we say above that a formula L "is equivalent to" a formula R, it is more than just saying that they have the same value in all models. Indeed, we are actually stating that the formula '((L→R)&(R→L))' is provable from our four logical axiom schemas of quantification. As noted, all the above equivalences require that ψ does not have x as a free variable name. If

ψ does happen to have x as a free variable name, then we can easily change the quantification to be over some other variable name, since '$\forall x[\phi(x)]$' is equivalent to '$\forall z[\phi(z)]$' (and similarly, '$\exists x[\phi(x)]$' is equivalent to '$\exists z[\phi(z)]$') for every variable name z that does not appear in ϕ. We will thus need two additional maneuvers, that handle not only replacing a variable name in a single formula (i.e., that '$Qx[\phi(x)]$' is equivalent to '$Qz[\phi(z)]$'), but also quantifying (with and without replacing variable names) over equivalent formulas. Specifically, let x and y be variable names, and let $\phi(\square)$ and $\psi(\square)$ be arbitrary parametrized formulas that have neither x nor y as free variable names. Then we will need the following:

15. If $\phi(x)$ and $\psi(x)$ are equivalent, then '$\forall x[\phi(x)]$' and '$\forall y[\psi(y)]$' are equivalent.
16. If $\phi(x)$ and $\psi(x)$ are equivalent, then '$\exists x[\phi(x)]$' and '$\exists y[\psi(y)]$' are equivalent.

In this section, you will prove the Prenex Normal Form Theorem. Our main strategy will be, as outlined above, to first make sure that no two variable names "clash," and then "pull out" all quantifications using the above equivalences. Let us consider an example: say that we wish to convert the following formula into prenex normal form:

'~(z=x|∀z[(∃x[(x=z&∀z[z=x])]→∀x[x=y])])'

Let us carefully try and make sense of all of the occurrences of the same variables names (variable name occurrences left in black are free):

'~(z=x|∀z[(∃x[(x=z&∀z[z=x])]→∀x[x=y])])'

We first replace each quantified variable name with a new unique variable name:

'~(z=x|∀z1[(∃z2[(z2=z1&∀z3[z3=z2])]→∀z4[z4=y])])'

We now start by pulling out the quantification '∀z3' from the conjunction:

'~(z=x|∀z1[(∃z2[(z2=z1&∀z3[z3=z2])]→∀z4[z4=y])])'
⇓
'~(z=x|∀z1[(∃z2[∀z3[(z2=z1&z3=z2)]]→∀z4[z4=y])])'

We now pull out both of the quantifications '∃z2' and '∀z3' from the implication (note that the universal quantification becomes an existential quantification and *vise versa*, due to the rules of pulling out quantifications from the left side of an implication):

'~(z=x|∀z1[(∃z2[∀z3[(z2=z1&z3=z2)]]→∀z4[z4=y])])'
⇓
'~(z=x|∀z1[∀z2[∃z3[((z2=z1&z3=z2)→∀z4[z4=y])]]])'

We now pull out the quantification '∀z4' from the implication (note that this remains a universal quantification due to the rules of pulling out quantifications from the right side of an implication):

'~(z=x|∀z1[∀z2[∃z3[((z2=z1&z3=z2)→∀z4[z4=y])]]])'
⇓
'~(z=x|∀z1[∀z2[∃z3[∀z4[((z2=z1&z3=z2)→z4=y)]]]])'

We now pull out all of the quantifications from the disjunction:

‘~(z=x|∀z1[∀z2[∃z3[∀z4[((z2=z1&z3=z2)→z4=y)]]]])’
⇓
‘~∀z1[∀z2[∃z3[∀z4[(z=x|((z2=z1&z3=z2)→z4=y))]]]]’

Finally, we pull out all of the quantifications from the negation, noting that each universal quantifier is replaced with an existential quantifier and *vise versa*:

‘~∀z1[∀z2[∃z3[∀z4[(z=x|((z2=z1&z3=z2)→z4=y))]]]]’
⇓
‘∃z1[∃z2[∀z3[∃z4[~(z=x|((z2=z1&z3=z2)→z4=y))]]]]’

That’s it: we have reached an equivalent formula that is in prenex normal form. In the remainder of this section, you will write code that not only performs the above conversion for any formula, but also produces a proof of the equivalence of the original formula and the resulting formula, thus proving the Prenex Normal Form Theorem.

To make things easier, we will allow the proofs that you will program in the remainder of this section to use all of the sixteen above-stated equivalences and implications as additional axiom schemas, even though each of these is provable from our four basic axiom schemas of quantification. All of these additional axiom schemas are already defined for you in the file `predicates/prenex.py`.

predicates/prenex.py
```
#: Additional axioms of quantification for Predicate Logic.
ADDITIONAL_QUANTIFICATION_AXIOMS = (
    Schema(Formula.parse('((~Ax[R(x)]->Ex[~R(x)])&(Ex[~R(x)]->~Ax[R(x)]))'),
           {'x', 'R'}),
    Schema(Formula.parse('((~Ex[R(x)]->Ax[~R(x)])&(Ax[~R(x)]->~Ex[R(x)]))'),
           {'x', 'R'}),
    Schema(Formula.parse('(((Ax[R(x)]&Q())->Ax[(R(x)&Q())])&'
                         '(Ax[(R(x)&Q())]->(Ax[R(x)]&Q())))'), {'x', 'R', 'Q'}),
    Schema(Formula.parse('(((Ex[R(x)]&Q())->Ex[(R(x)&Q())])&'
                         '(Ex[(R(x)&Q())]->(Ex[R(x)]&Q())))'), {'x', 'R', 'Q'}),
    Schema(Formula.parse('(((Q()&Ax[R(x)])->Ax[(Q()&R(x))])&'
                         '(Ax[(Q()&R(x))]->(Q()&Ax[R(x)])))'), {'x', 'R', 'Q'}),
    Schema(Formula.parse('(((Q()&Ex[R(x)])->Ex[(Q()&R(x))])&'
                         '(Ex[(Q()&R(x))]->(Q()&Ex[R(x)])))'), {'x', 'R', 'Q'}),
    Schema(Formula.parse('(((Ax[R(x)]|Q())->Ax[(R(x)|Q())])&'
                         '(Ax[(R(x)|Q())]->(Ax[R(x)]|Q())))'), {'x', 'R', 'Q'}),
    Schema(Formula.parse('(((Ex[R(x)]|Q())->Ex[(R(x)|Q())])&'
                         '(Ex[(R(x)|Q())]->(Ex[R(x)]|Q())))'), {'x', 'R', 'Q'}),
    Schema(Formula.parse('(((Q()|Ax[R(x)])->Ax[(Q()|R(x))])&'
                         '(Ax[(Q()|R(x))]->(Q()|Ax[R(x)])))'), {'x', 'R', 'Q'}),
    Schema(Formula.parse('(((Q()|Ex[R(x)])->Ex[(Q()|R(x))])&'
                         '(Ex[(Q()|R(x))]->(Q()|Ex[R(x)])))'), {'x', 'R', 'Q'}),
    Schema(Formula.parse('(((Ax[R(x)]->Q())->Ex[(R(x)->Q())])&'
                         '(Ex[(R(x)->Q())]->(Ax[R(x)]->Q())))'), {'x', 'R', 'Q'}),
    Schema(Formula.parse('(((Ex[R(x)]->Q())->Ax[(R(x)->Q())])&'
                         '(Ax[(R(x)->Q())]->(Ex[R(x)]->Q())))'), {'x', 'R', 'Q'}),
    Schema(Formula.parse('(((Q()->Ax[R(x)])->Ax[(Q()->R(x))])&'
                         '(Ax[(Q()->R(x))]->(Q()->Ax[R(x)])))'), {'x', 'R', 'Q'}),
    Schema(Formula.parse('(((Q()->Ex[R(x)])->Ex[(Q()->R(x))])&'
                         '(Ex[(Q()->R(x))]->(Q()->Ex[R(x)])))'), {'x', 'R', 'Q'}),
    Schema(Formula.parse('(((R(x)->Q(x))&(Q(x)->R(x)))->'
                         '((Ax[R(x)]->Ay[Q(y)])&(Ay[Q(y)]->Ax[R(x)])))'),
           {'x', 'y', 'R', 'Q'}),
    Schema(Formula.parse('(((R(x)->Q(x))&(Q(x)->R(x)))->'
                    '((Ex[R(x)]->Ey[Q(y)])&(Ey[Q(y)]->Ex[R(x)]))'),
        {'x', 'y', 'R', 'Q'}))
```

Notice that in the example above, we have in fact used these equivalences to replace a *sub*formula with an equivalent subformula. While the schemas above do not allow us to do precisely that (they formally handle full rather than partial formulas), as we will see the additional ability of Additional Axioms 15 and 16 to quantify over equivalent formulas will allow us to proceed in essentially the same way nonetheless.

As we will handle a lot of equivalence-between-formulas proofs below, it will be convenient to use the following function, which we have already implemented for you and which takes two formulas and returns the formula that asserts their equivalence.

predicates/prenex.py

```
def equivalence_of(formula1: Formula, formula2: Formula) -> Formula:
    """States the equivalence of the two given formulas as a formula.

    Parameters:
        formula1: first of the formulas the equivalence of which is to be
            stated.
        formula2: second of the formulas the equivalence of which is to be
            stated.

    Returns:
        The formula '((`formula1`->`formula2`)&(`formula2`->`formula1`))'.
    """
    return Formula('&', Formula('->', formula1, formula2),
                   Formula('->', formula2, formula1))
```

Before we start proving the Prenex Normal Form Theorem using the above schemas, however, we invite you to prove (all instances of) the first of these schemas, Additional Axiom 1, yourself from the four basic axiom schemas of quantification (and to save yourself the work of proving the remaining fifteen schemas). Your proof will also demonstrate the incredible usefulness of the Deduction Theorem from the previous section.

Optional Task 4. In this task, you will prove, for every variable name x and for every arbitrary formula $\phi(x)$, that '~$\forall x[\phi(x)]$' is equivalent to '$\exists x[$~$\phi(x)]$'. In the first two parts below you will show both directions of the equivalence, and in the third you will tautologically infer the equivalence. The functions that you are asked to implement in this task are contained in the file `predicates/some_proofs.py`.

a. Prove, for every variable name x and for every arbitrary formula $\phi(x)$, that '(~$\exists x[$~$\phi(x)]\rightarrow\forall x[\phi(x)]$)' holds. That is, implement the missing code for the function `_prove_not_exists_not_implies_all(variable, formula)`, which takes x and $\phi(x)$, and returns a proof of '(~$\exists x[$~$\phi(x)]\rightarrow\forall x[\phi(x)]$)'.[6]

predicates/some_proofs.py

```
def _prove_not_exists_not_implies_all(variable: str, formula: Formula,
                                      print_as_proof_forms: bool = False) -> \
        Proof:
    """Proves that '(~E`variable`[~`formula`]->A`variable`[`formula`])'.

    Parameters:
```

[6]Once again, here and below, we allow the usage of all six basic logical axiom schemas so that you may easily use the `Prover` class in your solution.

```
        variable: variable name for the quantifications in the formula to be
            proven.
        formula: statement to be universally quantified, and whose negation is
            to be existentially quantified, in the formula to be proven.
        print_as_proof_forms: flag specifying whether the proof is to be printed
            in real time as it is being created.

    Returns:
        A valid proof of the above formula via `Prover.AXIOMS`.
    """
    assert is_variable(variable)
    # Optional Task 11.4a
```

Guidelines: Use the Deduction Theorem, that is, assume '$\sim\exists x[\sim\phi(x)]$' and prove '$\forall x[\phi(x)]$'. As your first step, use '$\sim\phi(x)\rightarrow\exists x[\sim\phi(x)]$'.

b. Prove, for every variable name x and for every arbitrary formula $\phi(x)$, that '$(\exists x[\sim\phi(x)]\rightarrow\sim\forall x[\phi(x)])$' holds. That is, implement the missing code for the function `_prove_exists_not_implies_not_all(variable, formula)`, which takes x and $\phi(x)$, and returns a proof of '$(\exists x[\sim\phi(x)]\rightarrow\sim\forall x[\phi(x)])$'.

predicates/some_proofs.py

```
def _prove_exists_not_implies_not_all(variable: str, formula: Formula,
                                      print_as_proof_forms: bool = False) -> \
        Proof:
    """Proves that '(E`variable`[~`formula`]->~A`variable`[`formula`])'.

    Parameters:
        variable: variable name for the quantifications in the formula to be
            proven.
        formula: statement to be universally quantified, and whose negation is
            to be existentially quantified, in the formula to be proven.
        print_as_proof_forms: flag specifying whether the proof is to be printed
            in real time as it is being created.

    Returns:
        A valid proof of the above formula via `Prover.AXIOMS`.
    """
    assert is_variable(variable)
    # Optional Task 11.4b
```

Guidelines: Use the Deduction Theorem, that is, assume '$\exists x[\sim\phi(x)]$' and prove '$\sim\forall x[\phi(x)]$'. Your last step should be calling `add_existential_derivation()` with the `consequent` argument to this method being '$\sim\forall x[\phi(x)]$' and with the `line_number1` argument to this method pointing to a line with the assumption '$\exists x[\sim\phi(x)]$'.

c. Prove, for every variable name x and for every arbitrary formula $\phi(x)$, that '$\sim\forall x[\phi(x)]$' is equivalent to '$\exists x[\sim\phi(x)]$'. That is, implement the missing code for the function `prove_not_all_iff_exists_not(variable, formula)`, which takes x and $\phi(x)$, and returns a proof of this equivalence.

predicates/some_proofs.py

```
def prove_not_all_iff_exists_not(variable: str, formula: Formula,
                                 print_as_proof_forms: bool = False) -> Proof:
    """Proves that
    `equivalence_of('(~A`variable`[`formula`]', 'E`variable`[~`formula`]')`.

    Parameters:
        variable: variable name for the quantifications in the formula to be
            proven.
        formula: statement to be universally quantified, and whose negation is
            to be existentially quantified, in the formula to be proven.
        print_as_proof_forms: flag specifying whether the proof is to be printed
            in real time as it is being created.

    Returns:
        A valid proof of the above formula via `Prover.AXIOMS`.
    """
    assert is_variable(variable)
    # Optional Task 11.4c
```

Guidelines: Tautologically infer this from the previous two parts. Use the method `add_proof()` of class `Prover` to insert these two proofs into a new prover (the `add_proof()` method takes care of properly shifting all line numbers that justify MP and UG lines, to retain the validity of proof lines that are added when the prover already has some proof lines), and continue the proof from there using the `Prover` API.

Examining the proof of equivalence that Optional Task 4 outputs, you will note that it uses all four basic logical axioms of quantifications (UI, EI, US, and ES). We note that your solution to Optional Task 4 indeed completely proves (all instances of) the first of the sixteen schemas above. Therefore, in any proof that has this schema as an assumption/axiom (recall that in Python, this schema is referenced as `ADDITIONAL_QUANTIFICATION_AXIOMS[0]`), any step that instantiates this assumption/axiom, i.e., any proof step of the following form:

```
stepN = prover.add_instantiated_assumption(
            instance, ADDITIONAL_QUANTIFICATION_AXIOMS[0],
            {'R': formula, 'x': variable})
```

can be replaced with the following equivalent proof step, which does not require `ADDITIONAL_QUANTIFICATION_AXIOMS[0]` as an assumption/axiom:

```
stepN = prover.add_proof(
            not_all_iff_exists_not_proof(
                variable, formula.substitute({'_': Term(variable)})))
```

Now that we have had a small taste of what proving the above sixteen schemas entails, we will very happily move on to proving the Prenex Normal Form Theorem, using *all* sixteen schemas as additional assumptions/axioms. (The fact that all of these sixteen schemas are provable from our four basic axiom schemas of quantification implies, by the Soundness Theorem for Predicate Logic, that they are all sound. Since you have not seen a proof of Additional Axioms 2 through 16, we invite you to try and prove their soundness directly to be convinced that they are all indeed sound.) We start by making sure that the quantified variable names in different parts of a given formula do not “clash” with one another, nor with the free variable names of the formula.

Definition (Uniquely Named Variables)**.** A formula is said to have **uniquely named variables** if no two quantifications in the formula quantify over the same variable name, and no variable name has both bound and free occurrences in the formula.

The function `has_uniquely_named_variables(formula)`, which we have already implemented for you, checks whether a given formula has uniquely named variables.

predicates/prenex.py

```
def has_uniquely_named_variables(formula: Formula) -> bool:
    """Checks if the given formula has uniquely named variables.

    Parameters:
        formula: formula to check.

    Returns:
        ``False`` if in the given formula some variable name has both bound and
        free occurrences or is quantified by more than one quantifier, ``True``
        otherwise.

    Examples:
        >>> has_uniquely_named_variables(
        ...     Formula.parse('(x=0&(Ax[R(x)]|Ex[R(x)]))'))
        False
        >>> has_uniquely_named_variables(
        ...     Formula.parse('(x=0&(Ax[R(x)]|Ey[R(y)]))'))
        False
        >>> has_uniquely_named_variables(
        ...     Formula.parse('(y=0&(Ax[R(x)]|Ex[R(x)]))'))
        False
        >>> has_uniquely_named_variables(
        ...     Formula.parse('(x=0&(Ay[R(y)]|Ez[R(z)]))'))
        True
    """
    forbidden_variables = set(formula.free_variables())
    def has_uniquely_named_variables_helper(formula: Formula) -> bool:
        if is_unary(formula.root):
            return has_uniquely_named_variables_helper(formula.first)
        elif is_binary(formula.root):
            return has_uniquely_named_variables_helper(formula.first) and \
                   has_uniquely_named_variables_helper(formula.second)
        elif is_quantifier(formula.root):
            if formula.variable in forbidden_variables:
                return False
            forbidden_variables.add(formula.variable)
            return has_uniquely_named_variables_helper(formula.statement)
        else:
            assert is_equality(formula.root) or is_relation(formula.root)
            return True

    return has_uniquely_named_variables_helper(formula)
```

Task 5. Implement the missing code for the function `_uniquely_rename_quantified_variables(formula)`, which takes a formula and returns an equivalent formula with uniquely named variables, along with a proof of the equivalence of the given and returned formulas.

predicates/prenex.py

```
def _uniquely_rename_quantified_variables(formula: Formula) -> \
        Tuple[Formula, Proof]:
    """Converts the given formula to an equivalent formula with uniquely named
    variables, and proves the equivalence of these two formulas.

    Parameters:
        formula: formula to convert, which contains no variable names starting
            with ``z``.

    Returns:
        A pair. The first element of the pair is a formula equivalent to the
        given formula, with the exact same structure but with the additional
        property of having uniquely named variables, obtained by consistently
        replacing each variable name that is bound in the given formula with a
        new variable name obtained by calling
        `next(fresh_variable_name_generator)`. The second element of the pair is
        a proof of the equivalence of the given formula and the returned
        formula (i.e., a proof of `equivalence_of(formula, returned_formula)`)
        via `Prover.AXIOMS` and `ADDITIONAL_QUANTIFICATION_AXIOMS`.

    Examples:
        >>> formula = Formula.parse('~(w=x|Aw[(Ex[(x=w&Aw[w=x])]->Ax[x=y])])')
        >>> returned_formula, proof = _uniquely_rename_quantified_variables(
        ...     formula)
        >>> returned_formula
        ~(w=x|Az58[(Ez17[(z17=z58&Az4[z4=z17])]->Az32[z32=y])])
        >>> proof.is_valid()
        True
        >>> proof.conclusion == equivalence_of(formula, returned_formula)
        True
        >>> proof.assumptions == Prover.AXIOMS.union(
        ...     ADDITIONAL_QUANTIFICATION_AXIOMS)
        True
    """
    for variable in formula.variables():
        assert variable[0] != 'z'
    # Task 11.5
```

Guidelines: Use recursion. To modify, combine, and extend the proofs returned by recursive calls, construct a new `Prover` object, use the `add_proof()` method of that object with each proof returned by a recursive call to insert it into the proof of this prover (the `add_proof()` method takes care of properly shifting all line numbers that justify MP and UG lines, to retain the validity of proof lines that are added when the prover already has some proof lines), and then complete the proof using the `Prover` API, including the methods that you implemented in Chapter 10 (in particular, with the last two additional axioms listed above—Additional Axioms 15 and 16).

Recall that Additional Axioms 1 and 2 listed above allow you to "pull out" a single quantification across a negation. The next building block toward proving the Prenex Normal Form Theorem is to show how to use that (in conjunction with Additional Axioms 15 and 16) to "pull out" any number of quantifications across a negation.

Task 6. Implement the missing code for the function `_pull_out_quantifications_across_negation(formula)`, which takes a formula whose root is a negation, i.e., a formula of the form '$\sim Q_1x_1[Q_2x_2[\cdots Q_nx_n[\phi]\cdots]]$' where

$n \geq 0$, each Q_i is a quantifier, each x_i is a variable name, and ϕ does not start with a quantifier, and returns an equivalent formula of the form '$Q'_1x_1[Q'_2x_2[\cdots Q'_nx_n[\sim\phi]\cdots]]$' where each Q'_i is a quantifier (and where each x_i and ϕ are the same as in `formula`), along with a proof of the equivalence of the given and returned formulas.

predicates/prenex.py

```
def _pull_out_quantifications_across_negation(formula: Formula) -> \
        Tuple[Formula, Proof]:
    """Converts the given formula with uniquely named variables of the form
    '~`Q1``x1`[`Q2``x2`[...`Qn``xn`[`inner_formula`]...]]' to an equivalent
     formula of the form
    '`Q'1``x1`[`Q'2``x2`[...`Q'n``xn`[~`inner_formula`]...]]', and proves the
    equivalence of these two formulas.

    Parameters:
        formula: formula to convert, whose root is a negation, i.e., which is of
            the form '~`Q1``x1`[`Q2``x2`[...`Qn``xn`[`inner_formula`]...]]'
            where `n`>=0, each `Qi` is a quantifier, each `xi` is a variable
            name, and `inner_formula` does not start with a quantifier.

    Returns:
        A pair. The first element of the pair is a formula equivalent to the
        given formula, but of the form
        '`Q'1``x1`[`Q'2``x2`[...`Q'n``xn`[~`inner_formula`]...]]' where each
        `Q'i` is a quantifier, and where the `xi` variable names and
        `inner_formula` are the same as in the given formula. The second element
        of the pair is a proof of the equivalence of the given formula and the
        returned formula (i.e., a proof of
        `equivalence_of(formula, returned_formula)`) via `Prover.AXIOMS` and
        `ADDITIONAL_QUANTIFICATION_AXIOMS`.

    Examples:
        >>> formula = Formula.parse('~Ax[Ey[R(x,y)]]')
        >>> returned_formula, proof = _pull_out_quantifications_across_negation(
        ...     formula)
        >>> returned_formula
        Ex[Ay[~R(x,y)]]
        >>> proof.is_valid()
        True
        >>> proof.conclusion == equivalence_of(formula, returned_formula)
        True
        >>> proof.assumptions == Prover.AXIOMS.union(
        ...     ADDITIONAL_QUANTIFICATION_AXIOMS)
        True
    """
    assert is_unary(formula.root)
    # Task 11.6
```

Guidelines: Call the function recursively with '$\sim Q_2x_2[\cdots Q_nx_n[\phi]\cdots]$' to obtain '$Q'_2x_2[\cdots Q'_nx_n[\sim\phi]\cdots]$' (and the proof of equivalence). From this show that '$Q'_1x_1[\sim Q_2x_2[\cdots Q_nx_n[\phi]\cdots]]$' and '$Q'_1x_1[Q'_2x_2[\cdots Q'_nx_n[\sim\phi]\cdots]]$' are equivalent (don't forget that `add_proof()` is your friend); now apply Additional Axiom 1 or Additional Axiom 2. As a base case for the recursion, use the case $n = 0$ (no quantifications after the negation at the root of formula) rather than the case $n = 1$.

Our next building block, which you will develop throughout Tasks 7 and 8, is to "pull out" any number of quantifications from both operands of a binary operator. We start with "pulling out" any number of quantifications from one of the operands of a binary operator.

Task 7.

a. Implement the missing code for the function `_pull_out_quantifications_from_left_across_binary_operator(formula)`, which takes a formula with uniquely named variables whose root is a binary operator, i.e., a formula with uniquely named variables that is of the form '$(Q_1x_1[Q_2x_2[\cdots Q_nx_n[\phi]\cdots]]{*}\psi)$' where $*$ is a binary operator, $n \geq 0$, each Q_i is a quantifier, each x_i is a variable name, and ϕ does not start with a quantifier, and returns an equivalent formula of the form '$Q'_1x_1[Q'_2x_2[\cdots Q'_nx_n[(\phi{*}\psi)]\cdots]]$' where each Q'_i is a quantifier (and where $*$, each x_i, ϕ, and ψ are the same as in `formula`), along with a proof of the equivalence of the given and returned formulas.

predicates/prenex.py
```
def _pull_out_quantifications_from_left_across_binary_operator(formula:
                                                               Formula) -> \
        Tuple[Formula, Proof]:
    """Converts the given formula with uniquely named variables of the form
    '(`Q1``x1`[`Q2``x2`[...`Qn``xn`[`inner_first`]...]]`*``second`)' to an
    equivalent formula of the form
    '`Q'1``x1`[`Q'2``x2`[...`Q'n``xn`[(`inner_first``*``second`)]...]]' and
    proves the equivalence of these two formulas.

    Parameters:
        formula: formula with uniquely named variables to convert, whose root
            is a binary operator, i.e., which is of the form
            '(`Q1``x1`[`Q2``x2`[...`Qn``xn`[`inner_first`]...]]`*``second`)'
            where `*` is a binary operator, `n`>=0, each `Qi` is a quantifier,
            each `xi` is a variable name, and `inner_first` does not start with
            a quantifier.

    Returns:
        A pair. The first element of the pair is a formula equivalent to the
        given formula, but of the form
        '`Q'1``x1`[`Q'2``x2`[...`Q'n``xn`[(`inner_first``*``second`)]...]]'
        where each `Q'i` is a quantifier, and where the operator `*`, the `xi`
        variable names, `inner_first`, and `second` are the same as in the given
        formula. The second element of the pair is a proof of the equivalence of
        the given formula and the returned formula (i.e., a proof of
        `equivalence_of(formula, returned_formula)`) via `Prover.AXIOMS` and
        `ADDITIONAL_QUANTIFICATION_AXIOMS`.

    Examples:
        >>> formula = Formula.parse('(Ax[Ey[R(x,y)]]&Ez[P(1,z)])')
        >>> returned_formula, proof = \
        ...     _pull_out_quantifications_from_left_across_binary_operator(
        ...         formula)
        >>> returned_formula
        Ax[Ey[(R(x,y)&Ez[P(1,z)])]]
        >>> proof.is_valid()
        True
        >>> proof.conclusion == equivalence_of(formula, returned_formula)
        True
        >>> proof.assumptions == Prover.AXIOMS.union(
        ...     ADDITIONAL_QUANTIFICATION_AXIOMS)
        True
    """
    assert has_uniquely_named_variables(formula)
    assert is_binary(formula.root)
    # Task 11.7a
```

Guidelines: The logic is almost identical to that of Task 6: call the function recursively with '$(Q_2x_2[\cdots Q_nx_n[\phi]\cdots]*\psi)$' to obtain '$Q'_2x_2[\cdots Q'_nx_n[(\phi*\psi)]\cdots]$' (and the proof of equivalence). From this show that '$Q'_1x_1[(Q_2x_2[\cdots Q_nx_n[\phi]\cdots]*\psi)]$' and '$Q'_1x_1[Q'_2x_2[\cdots Q'_nx_n[(\phi*\psi)]\cdots]]$' are equivalent (don't forget that `add_proof()` is your friend); now apply one of the additional quantification axioms. As a base case for the recursion, use the case $n = 0$ rather than the case $n = 1$.

Hint: Q'_i depends not only on Q_i, but also on the operator $*$.

b. Implement the missing code for the function `_pull_out_quantifications_from_right_across_binary_operator(formula)`, which takes a formula with uniquely named variables whose root is a binary operator, i.e., a formula with uniquely named variables that is of the form '$(\phi*Q_1x_1[Q_2x_2[\cdots Q_nx_n[\psi]\cdots]])$' where $*$ is a binary operator, $n \geq 0$, each Q_i is a quantifier, each x_i is a variable name, and ψ does not start with a quantifier, and returns an equivalent formula of the form '$Q'_1x_1[Q'_2x_2[\cdots Q'_nx_n[(\phi*\psi)]\cdots]]$' where each Q'_i is a quantifier (and where $*$, each x_i, ϕ, and ψ are the same as in `formula`), along with a proof of the equivalence of the given and returned formulas.

predicates/prenex.py

```
def _pull_out_quantifications_from_right_across_binary_operator(formula:
                                                                Formula) -> \
        Tuple[Formula, Proof]:
    """Converts the given formula with uniquely named variables of the form
    '(`first``*``Q1``x1`[`Q2``x2`[...`Qn``xn`[`inner_second`]...]])' to an
    equivalent formula of the form
    '`Q'1``x1`[`Q'2``x2`[...`Q'n``xn`[(`first``*``inner_second`)]...]]' and
    proves the equivalence of these two formulas.

    Parameters:
        formula: formula with uniquely named variables to convert, whose root
            is a binary operator, i.e., which is of the form
            '(`first``*``Q1``x1`[`Q2``x2`[...`Qn``xn`[`inner_second`]...]])'
            where `*` is a binary operator, `n`>=0, each `Qi` is a quantifier,
            each `xi` is a variable name, and `inner_second` does not start with
            a quantifier.

    Returns:
        A pair. The first element of the pair is a formula equivalent to the
        given formula, but of the form
        '`Q'1``x1`[`Q'2``x2`[...`Q'n``xn`[(`first``*``inner_second`)]...]]'
        where each `Q'i` is a quantifier, and where the operator `*`, the `xi`
        variable names, `first`, and `inner_second` are the same as in the given
        formula. The second element of the pair is a proof of the equivalence of
        the given formula and the returned formula (i.e., a proof of
        `equivalence_of(formula, returned_formula)`) via `Prover.AXIOMS` and
        `ADDITIONAL_QUANTIFICATION_AXIOMS`.

    Examples:
        >>> formula = Formula.parse('(Ax[Ey[R(x,y)]]|Ez[P(1,z)])')
        >>> returned_formula, proof = \
        ...     _pull_out_quantifications_from_right_across_binary_operator(
        ...         formula)
        >>> returned_formula
        Ez[(Ax[Ey[R(x,y)]]|P(1,z))]
        >>> proof.is_valid()
        True
        >>> proof.conclusion == equivalence_of(formula, returned_formula)
        True
        >>> proof.assumptions == Prover.AXIOMS.union(
        ...     ADDITIONAL_QUANTIFICATION_AXIOMS)
        True
    """
    assert has_uniquely_named_variables(formula)
    assert is_binary(formula.root)
    # Task 11.7b
```

Guidelines: Almost identical to the first part: call the function recursively with '$(\phi{*}Q_2x_2[\cdots Q_nx_n[\psi]\cdots])$' to obtain '$Q'_2x_2[\cdots Q'_nx_n[(\phi{*}\psi)]\cdots]$' (and the proof of equivalence). From this show that '$Q'_1x_1[(\phi{*}Q_2x_2[\cdots Q_nx_n[\psi]\cdots])]$' and '$Q'_1x_1[Q'_2x_2[\cdots Q'_nx_n[(\phi{*}\psi)]\cdots]]$' are equivalent (don't forget that `add_proof()` is your friend); now apply one of the additional quantification axioms. As a base case for the recursion, use the case $n = 0$ rather than the case $n = 1$.

You are now ready to complete the building block that "pulls out" all quantifications from *both* operands of a binary operator—this turns out to be slightly trickier than simply applying both parts of Task 7.

Task 8. Implement the missing code for the function `_pull_out_quantifications_across_binary_operator(formula)`, which takes a formula with uniquely named variables whose root is a binary operator, i.e., a formula with uniquely named variables that is of the form

$$\text{`}(Q_1x_1[Q_2x_2[\cdots Q_nx_n[\phi]\cdots]]{*}P_1y_1[P_2y_2[\cdots P_my_m[\psi]\cdots]])\text{'}$$

where $*$ is a binary operator, $n \geq 0$, $m \geq 0$, each Q_i and each P_i is a quantifier, each x_i and each y_i is a variable name, and neither ϕ nor ψ starts with a quantifier, and returns an equivalent formula of the form

$$\text{`}Q'_1x_1[Q'_2x_2[\cdots Q'_nx_n[P'_1y_1[P'_2x_2[\cdots P'_my_m[(\phi{*}\psi)]\cdots]]]\cdots]]\text{'}$$

where each Q'_i and each P'_i is a quantifier (and where $*$, each x_i, each y_i, ϕ, and ψ are the same as in `formula`), along with a proof of the equivalence of the given and returned formulas.

predicates/prenex.py

```
def _pull_out_quantifications_across_binary_operator(formula: Formula) -> \
        Tuple[Formula, Proof]:
    """Converts the given formula with uniquely named variables of the form
    '(`Q1``x1`[`Q2``x2`[...`Qn``xn`[`inner_first`]...]]`*`
      `P1``y1`[`P2``y2`[...`Pm``ym`[`inner_second`]...]])'
    to an equivalent formula of the form
    '`Q'1``x1`[`Q'2``x2`[...`Q'n``xn`[
         `P'1``y1`[`P'2``y2`[...`P'm``ym`[(`inner_first``*``inner_second`)]...]]
     ]...]]'
    and proves the equivalence of these two formulas.
    Parameters:
        formula: formula with uniquely named variables to convert, whose root
            is a binary operator, i.e., which is of the form
            '(`Q1``x1`[`Q2``x2`[...`Qn``xn`[`inner_first`]...]]`*`
              `P1``y1`[`P2``y2`[...`Pm``ym`[`inner_second`]...]])'
            where `*` is a binary operator, `n`>=0, `m`>=0, each `Qi` and `Pi`
            is a quantifier, each `xi` and `yi` is a variable name, and neither
            `inner_first` nor `inner_second` starts with a quantifier.

    Returns:
        A pair. The first element of the pair is a formula equivalent to the
        given formula, but of the form
        '`Q'1``x1`[`Q'2``x2`[...`Q'n``xn`[
             `P'1``y1`[`P'2``y2`[...`P'm``ym`[
                 (`inner_first``*``inner_second`)
             ]...]]
         ]...]]'
        where each `Q'i` and `P'i` is a quantifier, and where the operator `*`,
        the `xi` and `yi` variable names, `inner_first`, and `inner_second` are
        the same as in the given formula. The second element of the pair is a
        proof of the equivalence of the given formula and the returned formula
        (i.e., a proof of `equivalence_of(formula, returned_formula)`) via
        `Prover.AXIOMS` and `ADDITIONAL_QUANTIFICATION_AXIOMS`.

    Examples:
        >>> formula = Formula.parse('(Ax[Ey[R(x,y)]]->Ez[P(1,z)])')
        >>> returned_formula, proof = \
        ...     _pull_out_quantifications_across_binary_operator(
        ...         formula)
        >>> returned_formula
        Ex[Ay[Ez[(R(x,y)->P(1,z))]]]
        >>> proof.is_valid()
        True
        >>> proof.conclusion == equivalence_of(formula, returned_formula)
        True
        >>> proof.assumptions == Prover.AXIOMS.union(
        ...     ADDITIONAL_QUANTIFICATION_AXIOMS)
        True
    """
    assert has_uniquely_named_variables(formula)
    assert is_binary(formula.root)
    # Task 11.8
```

Guidelines: First use the first part of Task 7 on `formula` to obtain '$Q'_1x_1[Q'_2x_2[\cdots Q'_nx_n[(\phi{*}P_1y_1[P_2y_2[\cdots P_my_m[\psi]\cdots]])]\cdots]]$' (and the proof of equivalence). Then use the second part on '$(\phi{*}P_1y_1[P_2y_2[\cdots P_my_m[\psi]\cdots]])$' to obtain '$P'_1y_1[P'_2y_2[\cdots P'_my_m[(\phi{*}\psi)]\cdots]]$' (and the proof of equivalence). Use the latter to show that '$Q'_1x_1[Q'_2x_2[\cdots Q'_nx_n[(\phi{*}P_1y_1[P_2y_2[\cdots P_my_m[\psi]\cdots]])]\cdots]]$' is equivalent to $Q'_1x_1[Q'_2x_2[\cdots Q'_nx_n[P'_1y_1[P'_2x_2[\cdots P'_my_m[(\phi{*}\psi)]\cdots]]]\cdots]]$. (And don't forget that `add_proof()` is your friend.)

You are now ready to combine Tasks 6 and 8 to convert a formula with no "clashing" variable names (as output by Task 5) into a formula in prenex normal form.

Task 9. Implement the missing code for the function `_to_prenex_normal_form_from_uniquely_named_variables(formula)`, which takes a formula with uniquely named variables and returns an equivalent formula in prenex normal form, along with a proof of the equivalence of the given and returned formulas.

predicates/prenex.py

```
def _to_prenex_normal_form_from_uniquely_named_variables(formula: Formula) -> \
        Tuple[Formula, Proof]:
    """Converts the given formula with uniquely named variables to an equivalent
    formula in prenex normal form, and proves the equivalence of these two
    formulas.

    Parameters:
        formula: formula with uniquely named variables to convert.

    Returns:
        A pair. The first element of the pair is a formula equivalent to the
        given formula, but in prenex normal form. The second element of the pair
        is a proof of the equivalence of the given formula and the returned
        formula (i.e., a proof of `equivalence_of(formula, returned_formula)`)
        via `Prover.AXIOMS` and `ADDITIONAL_QUANTIFICATION_AXIOMS`.

    Examples:
        >>> formula = Formula.parse('(~(Ax[Ey[R(x,y)]]->Ez[P(1,z)])|S(w))')
        >>> returned_formula, proof = \
        ...     _to_prenex_normal_form_from_uniquely_named_variables(
        ...         formula)
        >>> returned_formula
        Ax[Ey[Az[(~(R(x,y)->P(1,z))|S(w))]]]
        >>> proof.is_valid()
        True
        >>> proof.conclusion == equivalence_of(formula, returned_formula)
        True
        >>> proof.assumptions == Prover.AXIOMS.union(
        ...     ADDITIONAL_QUANTIFICATION_AXIOMS)
        True
    """
    assert has_uniquely_named_variables(formula)
    # Task 11.9
```

Guidelines: In the cases in which the root of `formula` is an operator, use recursion to convert each operand into prenex normal form (and to obtain the proof of equivalence); complete the proof of each of these cases using Task 6 or Task 8. (Don't forget that `add_proof()` is your friend.)

Now that all of the recursions are behind you, you are finally ready to prove the Prenex Normal Form Theorem.

Task 10 (Programmatic Proof of the Prenex Normal Form Theorem)**.** Implement the missing code for the function `to_prenex_normal_form(formula)`, which takes a formula and returns an equivalent formula in prenex normal form, along with a proof of the equivalence of the given and returned formulas.

predicates/prenex.py

```
def to_prenex_normal_form(formula: Formula) -> Tuple[Formula, Proof]:
    """Converts the given formula to an equivalent formula in prenex normal
    form, and proves the equivalence of these two formulas.

    Parameters:
        formula: formula to convert, which contains no variable names starting
            with ``z``.

    Returns:
        A pair. The first element of the pair is a formula equivalent to the
        given formula, but in prenex normal form. The second element of the pair
        is a proof of the equivalence of the given formula and the returned
        formula (i.e., a proof of `equivalence_of(formula, returned_formula)`)
        via `Prover.AXIOMS` and `ADDITIONAL_QUANTIFICATION_AXIOMS`.

    Examples:
        >>> formula = Formula.parse('~(w=x|Aw[(Ex[(x=w&Aw[w=x])]->Ax[x=y])])')
        >>> returned_formula, proof = to_prenex_normal_form(formula)
        >>> returned_formula
        Ez58[Ez17[Az4[Ez32[~(w=x|((z17=z58&z4=z17)->z32=y))]]]]
        >>> proof.is_valid()
        True
        >>> proof.conclusion == equivalence_of(formula, returned_formula)
        True
        >>> proof.assumptions == Prover.AXIOMS.union(
        ...     ADDITIONAL_QUANTIFICATION_AXIOMS)
        True
    """
    for variable in formula.variables():
        assert variable[0] != 'z'
    # Task 11.10
```

Guidelines: Use Tasks 5 and 9 (and don't forget that `add_proof()` is your friend).