Chapter 10:
Working with First-Order Logic Proofs

In this chapter we will introduce a specific axiomatic system for first-order logic, and prove some theorems using this system. You will be asked both to prove things "manually" and to write some helper functions in Python that will make writing proofs easier (but no worries—you will not be asked to re-implement `inline_proof()` for first-order logic).

1 Our Axiomatic System

Our axiomatic system will of course have the following components that you have already dealt with in the `Proof` class in Chapter 9:

1. Inference Rules. As specified in Chapter 9, we have only two inference rules:
   - **Modus Ponens (MP):** From ϕ and '$(\phi\rightarrow\psi)$', deduce ψ. (Just like in propositional logic.)
   - **Universal Generalization (UG):** From ϕ deduce '$\forall x[\phi]$'. Note that ϕ may have x as a free variable (and may of course have any other free variable).
2. Tautologies (taken from propositional logic). As discussed in Chapter 9, we directly allow all tautologies as axioms purely for convenience, as it is also possible instead to leverage our prior analysis of propositional logic to prove them from the schema equivalents of the axioms of propositional logic.
   - **Tautology:** Any formula ϕ that is a tautology. Note that we allow tautologies to have free variables, such as in the tautology '((R(x)&Q(x))→R(x))'.

In addition to the above, our axiomatic system will also have the following six additional axioms schemas that deal with quantification and equality, which will be part of the assumptions/axioms of every proof that you will write from this point onward. These schemas are defined as constants in the file `predicates/prover.py` inside the class `Prover`, which we will discuss below.

3. Quantification Axioms. These ensure that the quantification symbols have the meanings that we want.
   - **Universal Instantiation (UI):** the schema '$(\forall x[\phi(x)]\rightarrow\phi(\tau))$', where $\phi(\square)$, x, and τ are (placeholders for) a parametrized formula, a variable name, and a term respectively.

predicates/prover.py
```
class Prover:
        ⋮
    #: Axiom schema of universal instantiation
    UI = Schema(Formula.parse('(Ax[R(x)]->R(c))'), {'R', 'x', 'c'})
```

- **Existential Introduction (EI):** the schema '$(\phi(\tau)\rightarrow\exists x[\phi(x)])$', where $\phi(\square)$, x, and τ are a parametrized formula, a variable name, and a term respectively.

predicates/prover.py
```
class Prover:
        ⋮
    #: Axiom schema of existential introduction
    EI = Schema(Formula.parse('(R(c)->Ex[R(x)])'), {'R', 'x', 'c'})
```

- **Universal Simplification (US):** the schema '$(\forall x[(\phi\rightarrow\psi(x))]\rightarrow(\phi\rightarrow\forall x[\psi(x)]))$', where ϕ is a (parameter-less) formula, $\psi(\square)$ is a parametrized formula, and x is a variable name. Note that the rules from Chapter 9 that define the legal instances of schemas require in particular that (the formula that is substituted for) ϕ does not have (the variable that is substituted for) x as a free variable.

predicates/prover.py
```
class Prover:
        ⋮
    #: Axiom schema of universal simplification
    US = Schema(Formula.parse('(Ax[(Q()->R(x))]->(Q()->Ax[R(x)]))'),
                {'Q', 'R', 'x'})
```

- **Existential Simplification (ES):** the schema '$((\forall x[(\psi(x)\rightarrow\phi)]\&\exists x[\psi(x)])\rightarrow\phi)$', where ϕ is a formula, $\psi(\square)$ is a parametrized formula, and x is a variable name. Note once again that the rules from Chapter 9 that define the legal instances of schemas require in particular that ϕ does not have x as a free variable.

predicates/prover.py
```
class Prover:
        ⋮
    #: Axiom schema of existential simplification
    ES = Schema(Formula.parse('((Ax[(R(x)->Q())]&Ex[R(x)])->Q())'),
                {'Q', 'R', 'x'})
```

4. Equality Axioms. These ensure that the equality symbol has the meaning that we want.[1]

- **Reflexivity (RX):** the schema '$\tau=\tau$', where τ is a term.

predicates/prover.py
```
class Prover:
        ⋮
    #: Axiom schema of reflexivity
    RX = Schema(Formula.parse('c=c'), {'c'})
```

- **Meaning of Equality (ME):** the schema '$(\tau=\sigma\rightarrow(\phi(\tau)\rightarrow\phi(\sigma)))$', where $\phi(\square)$ is a parametrized formula, and τ and σ are terms.

predicates/prover.py
```
class Prover:
        ⋮
```

[1] It is instructive to compare these two equality axioms with the formulas that you created in Chapter 8 to capture the properties of the 'SAME' relation that you used to replace equality there. RX of course corresponds to the reflexivity property, and ME in particular also implies that all relations must respect equality. What about symmetry and transitivity, though? As it turns out—you will show this in Tasks 6 and 9 below—these can be deduced from RX and ME due to the fact that ME allows substitution in arbitrary formulas and not only in relations.

```
    #: Axiom schema of meaning of equality
    ME = Schema(Formula.parse('(c=d->(R(c)->R(d)))'), {'R', 'c', 'd'})
```

predicates/prover.py

```
class Prover:
        ⋮
    #: Axiomatic system for first-order predicate logic
    AXIOMS = frozenset({UI, EI, US, ES, RX, ME})
```

Our first order of business is to verify that the above axiom schemas are sound. While for our axioms of propositional logic (and for their schema equivalents for predicate logic) this was an easy finite check, for the above axiom schemas this is no longer the case, as even for a single instance of any of these schemas there are infinitely many possible models, and no clear "shortcut" for showing that only finitely many values in these models affect its value. We will thus have to resort to "standard" mathematical proofs for proving the soundness of these axiom schemas.[2]

Lemma. *The six axiom schemas UI, EI, US, ES, RX, and ME are sound.*

Proof. We will prove this for the schema UI. The proof for each of the other five axiom schemas is either similar or easier. By definition, we have to prove that every instance of the schema UI is sound. Our proof will intimately depend on the two rules from Chapter 9 that define the legal instances of schemas, and in fact, we have defined these two rules precisely to make sure that all legal instances of each of our six axiom schemas are sound (and that these schemas have rich enough legal instances for our proofs). Let '$(\forall\bar{x}[\psi]\rightarrow\xi)$' be an instance of UI obtained by instantiating UI by substituting some variable name $\bar{x}$ for the placeholder x in UI, some parametrized formula $\bar{\phi}(\Box)$ for the placeholder ϕ in UI, and some term $\bar{\tau}$ for the placeholder τ in UI.

We will first claim that if we take ψ and replace every free occurrence of $\bar{x}$ in it by $\bar{\tau}$, then we obtain ξ.[3] Indeed, by the first rule from Chapter 9 that defines the legal instances of schemas, the formula $\bar{\phi}(\Box)$ does not have any free occurrences of $\bar{x}$. Therefore, *every* free occurrence of $\bar{x}$ in ψ is replaced with $\bar{\tau}$ in ξ. Now, by the second rule from Chapter 9 that defines the legal instances of schemas, $\bar{x}$, when substituted into $\bar{\phi}(\Box)$, does not get bound by a quantifier in $\bar{\phi}(\Box)$. Therefore, *only* free occurrences of $\bar{x}$ in ψ are replaced with $\bar{\tau}$ in ξ. So, ξ is the result of replacing every free occurrence of $\bar{x}$ in ψ by $\bar{\tau}$.

We will now claim that whenever the instance '$(\forall\bar{x}[\psi]\rightarrow\xi)$' is evaluated under any model M and assignment A for it, the value of every variable occurrence in ξ that originates in $\bar{\tau}$ (when substituted into $\bar{\phi}(\Box)$) gets its values from A. Indeed, by the second rule from Chapter 9 that defines the legal instances of schemas, no such variable occurrence gets bound by a quantifier in $\bar{\phi}(\Box)$, and therefore is free in '$(\forall\bar{x}[\psi]\rightarrow\xi)$', and so gets its value from A.

[2] While these "standard" mathematical proofs will *convince you* that these axiom schemas are sound, we note that we do have a problem of circularity here: if we ever wanted to formalize these mathematical proofs (that prove the soundness of these axiom schemas) as `Proof` objects, we would in fact need these axiom schemas—whose soundness these proofs prove—for formalizing these proofs. This is precisely why axioms are needed in mathematics: to avoid such circularity, we must assume *something* without a proof. We nonetheless write a "standard" mathematics proof for the soundness of these axiom schemas (even though this proof will implicitly assume things about quantifications, equality, etc.), to convince ourselves that they are reasonable axiom schemas to assume.

[3] In fact, the traditional way to define UI is via such replacements rather than via parametrized formulas. We chose the latter so that our schema syntax may be more intuitive for programmers.

We are now ready to show that '$(\forall\bar{x}[\psi]\rightarrow\xi)$' sound, i.e., evaluates to *True* under any model M and assignment A for it. By definition of how the *implies* operator is evaluated, it is enough to show that for any such model M and assignment A in which '$\forall\bar{x}[\psi]$' evaluates to *True*, ξ also evaluates to *True*. So let M and A be such so that '$\forall\bar{x}[\psi]$' evaluates to *True*. Let ω be the element in the universe of M to which $\bar{\tau}$ evaluates according to M and A. As we have argued, each of the variables occurrences in ξ that originates in $\bar{\tau}$ gets its value from A when ξ is evaluated. Therefore, and since ξ is the result of replacing every free occurrence of $\bar{x}$ in ψ by $\bar{\tau}$, we have that the value of ξ in M and A is the value of ψ in M and in the assignment obtained by augmenting A with the assignment of the value ω to the variable $\bar{x}$. But this value is *True* (which is what we want to prove!) since by the definition of '$\forall\bar{x}[\psi]$' evaluating to *True* in M and A, we have that ψ evaluates to *True* in M and in any assignment obtained by augmenting A with the assignment of *any* variable in the universe of M to the variable $\bar{x}$. $\square$

Combining the above lemma with the Soundness Theorem for Predicate Logic, we therefore obtain that any inference proven from these six axiom schemas is sound.

We once again emphasize that we explicitly allow our proofs to use formulas that are not **sentences**, i.e., formulas that have free variables. As discussed in Chapter 9, this gives a convenient way to manipulate formulas using tautological rules.[4] However, notice that a formula $\phi(x_1,\ldots,x_k)$ whose free variables are $x_1,\ldots,x_k$ is essentially equivalent to the sentence '$\forall x_1[\forall x_2[\cdots\forall x_k[\phi(x_1,\ldots,x_k)]\cdots]]$'; indeed, the inference rule of Universal Generalization (UG) allows us to move from the former to the latter (when applied to x_k, then to x_{k-1}, etc.), while the axiom schema of Universal Instantiation (UI) allows us to move from the latter to the former (when instantiated with 'x_1' as τ, then with 'x_2' as τ, etc.).

In this chapter, you will write proofs using the above axiomatic system, by using the `Proof` class that you built in Chapter 9, with UI, EI, US, ES, RX, and ME as axioms (in addition to any assumptions) in every proof that you will write. In some of the tasks below, you are asked to write some proof; the corresponding programming task is to return an object of class `Proof` that has as axioms/assumptions the axiom schemas UI, EI, US, ES, RX, and ME, as well as the axioms/assumptions specified in that task, and has the specified conclusion (and is a valid proof, of course...). Manually writing these proofs may turn out to be a bit (OK, very) cumbersome, so you will not work directly with the `Proof` class, but rather with a new class called `Prover` that "wraps around" the `Proof` class and provides a more convenient way to write proofs.

predicates/prover.py

```
class Prover:
    """A class for gradually creating a first-order logic proof from given
    assumptions as well as from the six axioms (`AXIOMS`) Universal
    Instantiation (`UI`), Existential Introduction (`EI`), Universal
    Simplification (`US`), Existential Simplification (`ES`), Reflexivity
    (`RX`), and Meaning of Equality (`ME`).

    Attributes:
        _assumptions: the assumptions/axioms of the proof being created, which
            include `AXIOMS`.
```

[4] Recall that the **propositional skeleton** of any first-order formula with a quantification at its root, such as '∀x[(Q(x)→(P(x)→Q(x)))]', is simply a propositional atom, and therefore not a tautology, and so such formulas are not considered axiomatic in our system. On the other hand, the propositional skeleton of '(Q(x)→(P(x)→Q(x)))' is a tautology, and so it is an axiom of our system.

```
        _lines: the current lines of the proof being created.
        _print_as_proof_forms: flag specifying whether the proof being created
            is being printed in real time as it forms.
    """
    _assumptions: FrozenSet[Schema]
    _lines: List[Proof.Line]
    _print_as_proof_forms: bool

    ⋮
    def __init__(self, assumptions: Collection[Union[Schema, Formula, str]],
                 print_as_proof_forms: bool=False) -> None:
        """Initializes a `Prover` from its assumptions/additional axioms. The
        proof created by the prover initially has no lines.

        Parameters:
            assumptions: the assumptions/axioms beyond `AXIOMS` for the proof
                to be created, each specified as either a schema, a formula that
                constitutes the unique instance of the assumption, or the string
                representation of the unique instance of the assumption.
            print_as_proof_forms: flag specifying whether the proof is to be
                printed in real time as it forms.
        """
        self._assumptions = \
            Prover.AXIOMS.union(
                {assumption if isinstance(assumption, Schema)
                 else Schema(assumption) if isinstance(assumption, Formula)
                 else Schema(Formula.parse(assumption))
                 for assumption in assumptions})
        self._lines = []
        self._print_as_proof_forms = print_as_proof_forms
        if self._print_as_proof_forms:
            print('Proving from assumptions/axioms:\n'
                  '  AXIOMS')
            for assumption in self._assumptions - Prover.AXIOMS:
                  print('  ' + str(assumption))
            print('Lines:')
```

A single instance of class Prover is used to “write” a single proof that initially has no lines when the prover is constructed (and that the methods of the prover can be used to gradually extend). As can be seen, for your convenience the constructor of class Prover is very flexible with respect to the types of the arguments that it can receive: while Proof assumptions are schemas, they can be passed to the Prover constructor not only as objects of type Schema, but also as objects of type Formula and even as their string representations, and the Prover constructor will convert them to type Schema. This flexibility is also a feature of the other methods of class Prover. For example, the function add_instantiated_assumption(), which corresponds to adding an assumption line to the proof being constructed by the current prover, can take the added instance not only as a Formula object but also as its string representation, and can even have string representations instead of Formula and Term objects as values in the instantiation map that it takes. The basic methods of class Prover, which are already implemented for you, are add_assumption(), add_instantiated_assumption(), add_tautology(), add_mp(), and add_ug(). Each of these methods adds to the proof being created by the prover a single line justified by one of the four allowed justification types, and we will get to know them momentarily.[5]

In addition the the above basic methods that add a single line to the proof being created by the prover, some of the tasks below will ask you to implement more advanced methods of the `Prover` class, each of which will add to the proof being created several lines with a single call. **Important:** In all of these methods that you will implement, you will of course access the `_lines` instance variable of the current `Prover` object that holds the lines already added to the proof being created, but you should *never* modify this instance variable directly via `self._lines`, but *only* via the methods `add_assumption()`, `add_instantiated_assumption()`, `add_tautology()`, `add_mp()`, and `add_ug()` (or via other methods that you will have already implemented). By convention, each of the `Prover` methods (the basic ones and the additional ones that you will implement) should return the line number, in the proof being created by the current prover, of the last (or the only) line that was added—the line that holds the conclusion that the method was asked to deduce.

Once you are done adding all desired lines to a prover, you can obtain the final proof, as a object of class `Proof`, by calling the `qed()` method of the prover.

predicates/prover.py

```
class Prover:
        ⋮
    def qed(self) -> Proof:
        """Concludes the proof created by the current prover.

        Returns:
            A valid proof, from the assumptions of the current prover as well as
            `AXIOMS`', of the formula justified by the last line appended to the
            current prover.
        """
        conclusion = self._lines[-1].formula
        if self._print_as_proof_forms:
            print('Conclusion:', str(conclusion) + '. QED\n')
        return Proof(self._assumptions, conclusion, self._lines)
```

2 Syllogisms

From Wikipedia:

> *A syllogism (Greek: "conclusion, inference") is a kind of logical argument that applies deductive reasoning to arrive at a conclusion based on two or more propositions that are asserted or assumed to be true.*
>
> *In its earliest form, defined by Aristotle, from the combination of a general statement (the major premise) and a specific statement (the minor premise), a conclusion is deduced. For example, knowing that all men are mortal (major premise) and that Socrates is a man (minor premise), we may validly conclude that Socrates is mortal.*

Let us try to formalize and prove the above syllogism in our system.
Assumptions: (In addition to the six axiom schemas UI, EI, US, ES, RX, ME)

[5]You can ignore the additional already-implemented method `add_proof()` for this chapter—you will use this method, which is a predicate-logic equivalent of sorts of the `inline_proof()` function that you implemented for propositional logic, in Chapters 11 and 12.

1. '∀x[(Man(x)→Mortal(x))]'
2. 'Man(aristotle)'

Conclusion: 'Mortal(aristotle)'
Proof:

1. '∀x[(Man(x)→Mortal(x))]'. Justification: first assumption.
2. '(∀x[(Man(x)→Mortal(x))]→(Man(aristotle)→Mortal(aristotle)))'. Justification: UI with $\phi(\Box)$ defined as '(Man(□)→Mortal(□))', with x defined as 'x', and with τ defined as 'aristotle'.
3. '(Man(aristotle)→Mortal(aristotle))'. Justification: MP from Steps 1 and 2.
4. 'Man(aristotle)'. Justification: second assumption.
5. 'Mortal(aristotle)'. Justification: MP from Steps 4 and 3.

A programmatic implementation of the above proof (and of all other proofs from this chapter) using the `Prover` class can be found in a corresponding function in the file `predicates/some_proofs.py`.

predicates/some_proofs.py

```
def syllogism_proof(print_as_proof_forms: bool = False) -> Proof:
    """Proves from the assumptions:

    1. All men are mortal ('Ax[(Man(x)->Mortal(x))]'), and
    2. Aristotle is a man ('Man(aristotle)')

    the conclusion: Aristotle is mortal ('Mortal(aristotle)').

    Parameters:
        print_as_proof_forms: flag specifying whether the proof is to be printed
            in real time as it is being constructed.

    Returns:
        A valid proof of the above inference via `AXIOMS`.
    """
    prover = Prover({'Ax[(Man(x)->Mortal(x))]', 'Man(aristotle)'},
                    print_as_proof_forms)
    step1 = prover.add_assumption('Ax[(Man(x)->Mortal(x))]')
    step2 = prover.add_instantiated_assumption(
        '(Ax[(Man(x)->Mortal(x))]->(Man(aristotle)->Mortal(aristotle)))',
        Prover.UI, {'R': '(Man(_)->Mortal(_))', 'c': 'aristotle'})
    step3 = prover.add_mp('(Man(aristotle)->Mortal(aristotle))', step1, step2)
    step4 = prover.add_assumption('Man(aristotle)')
    step5 = prover.add_mp('Mortal(aristotle)', step4, step3)
    return prover.qed()
```

This is a good opportunity to start to get to know the `Prover` class by going over the above Python implementation and comparing it to the above proof. In both programming and text, it is a tad annoying to go through Step 2 to obtain Step 3, so in the next task you will write a helper method called `add_universal_instantiation()` that can do this automatically for you and allows for the following shorter implementation:

predicates/some_proofs.py
```
def syllogism_proof_with_universal_instantiation(print_as_proof_forms: bool =
                                                  False) -> Proof:
    """Using the method `Prover.add_universal_instantiation`, proves from the
    assumptions:

    1. All men are mortal ('Ax[(Man(x)->Mortal(x))]'), and
    2. Aristotle is a man ('Man(aristotle)')

    the conclusion: Aristotle is mortal ('Mortal(aristotle)').

    Parameters:
        print_as_proof_forms: flag specifying whether the proof is to be printed
            in real time as it is being constructed.

    Returns:
        A valid proof, constructed using the method
        `Prover.add_universal_instantiation`, of the above inference via
        `AXIOMS`.
    """
    prover = Prover({'Ax[(Man(x)->Mortal(x))]', 'Man(aristotle)'},
                    print_as_proof_forms)
    step1 = prover.add_assumption('Ax[(Man(x)->Mortal(x))]')
    step2 = prover.add_universal_instantiation(
        '(Man(aristotle)->Mortal(aristotle))', step1, 'aristotle')
    step3 = prover.add_assumption('Man(aristotle)')
    step4 = prover.add_mp('Mortal(aristotle)', step3, step2)
    return prover.qed()
```

Task 1. Implement the missing code for the method `add_universal_instantiation(instantiation, line_number, term)`, which adds to the current prover a sequence of validly justified lines, the last of which has the formula `instantiation`.[6] The line with the given line number must hold a universally quantified formula '$\forall x[\phi(x)]$' for some variable name x and formula ϕ, and the derived formula `instantiation` should have the given term substituted into the variable name x in ϕ.

predicates/prover.py
```
class Prover:
        ⋮
    def add_universal_instantiation(self, instantiation: Union[Formula, str],
                                    line_number: int, term: Union[Term, str]) \
            -> int:
        """Appends to the proof being created by the current prover a sequence
        of validly justified lines, the last of which validly justifies the
        given formula, which is the result of substituting a term for the
        outermost universally quantified variable name of the formula of the
        specified already existing line of the proof.

        Parameters:
            instantiation: conclusion of the sequence of lines to be appended,
                specified as either a formula or its string representation.
            line_number: line number in the proof of a universally quantified
```

[6] The `instantiation` parameter can also be passed as a string representation of a formula. The code of the method that is already implemented for you converts it to a `Formula` if needed. The same holds for all `Formula` and `Term` objects passed, whether directly as arguments or even indirectly as values of maps, to any of the `Prover` methods that you will be asked to implement below.

```
            formula of the form 'A`x`[`predicate`]'.
        term: term, specified as either a term or its string representation,
            that when substituted into the free occurrences of `x` in
            `predicate` yields the given formula.

    Returns:
        The line number of the newly appended line that justifies the given
        formula in the proof being created by the current prover.

    Examples:
        If Line `line_number` contains the formula
        'Ay[Az[f(x,y)=g(z,y)]]' and `term` is 'h(w)', then `instantiation`
        should be 'Az[f(x,h(w))=g(z,h(w))]'.
    """
    if isinstance(instantiation, str):
        instantiation = Formula.parse(instantiation)
    assert line_number < len(self._lines)
    quantified = self._lines[line_number].formula
    assert quantified.root == 'A'
    if isinstance(term, str):
        term = Term.parse(term)
    assert instantiation == \
           quantified.predicate.substitute({quantified.variable: term})
    # Task 10.1
```

Example: If we have in Line 17 of the proof being constructed by a prover `p` the formula '∀y[(Q(y,z)→R(w,y))]', then the call

`p.add_universal_instantiation('(Q(f(1,w),z)->R(w,f(1,w)))', 17, 'f(1,w)')`

will add to the proof a few lines, the last of which proves '(Q(f(1,w),z) →R(w,f(1,w)))'.

Let us try to formalize and prove another syllogism: All Greeks are human, all humans are mortal, and thus all Greeks are mortal.

Assumptions:

1. '∀x[(Greek(x)→Human(x))]'
2. '∀x[(Human(x)→Mortal(x))]'

Conclusion: '∀x[(Greek(x)→Mortal(x))]'

Proof:

1. '∀x[(Greek(x)→Human(x))]'. Justification: first assumption.
2. '(Greek(x)→Human(x))'. Justification: universal instantiation of Step 1 (substituting the term 'x' for the quantification variable 'x').
3. '∀x[(Human(x)→Mortal(x))]'. Justification: second assumption.
4. '(Human(x)→Mortal(x))'. Justification: universal instantiation of Step 3.
5. '((Greek(x)→Human(x))→((Human(x)→Mortal(x))→(Greek(x)→Mortal(x))))'. Justification: a tautology.
6. '((Human(x)→Mortal(x))→(Greek(x)→Mortal(x)))'. Justification: MP from Steps 2 and 5.

7. '(Greek(x)→Mortal(x))'. Justification: MP from Steps 4 and 6.

8. '∀x[(Greek(x)→Mortal(x))]'. Justification: UG of Step 7.

predicates/some_proofs.py
```
def syllogism_all_all_proof(print_as_proof_forms: bool = False) -> Proof:
    """Proves from the assumptions:

    1. All Greeks are human ('Ax[(Greek(x)->Human(x))]'), and
    2. All humans are mortal ('Ax[(Human(x)->Mortal(x))]')

    the conclusion: All Greeks are mortal ('Ax[(Greek(x)->Mortal(x))]').

    Parameters:
        print_as_proof_forms: flag specifying whether the proof is to be printed
            in real time as it is being constructed.

    Returns:
        A valid proof of the above inference via `AXIOMS`.
    """
    prover = Prover({'Ax[(Greek(x)->Human(x))]', 'Ax[(Human(x)->Mortal(x))]'},
                    print_as_proof_forms)
    step1 = prover.add_assumption('Ax[(Greek(x)->Human(x))]')
    step2 = prover.add_universal_instantiation(
        '(Greek(x)->Human(x))', step1, 'x')
    step3 = prover.add_assumption('Ax[(Human(x)->Mortal(x))]')
    step4 = prover.add_universal_instantiation(
        '(Human(x)->Mortal(x))', step3, 'x')
    step5 = prover.add_tautology(
        '((Greek(x)->Human(x))->((Human(x)->Mortal(x))->(Greek(x)->Mortal(x))))')
    step6 = prover.add_mp(
        '((Human(x)->Mortal(x))->(Greek(x)->Mortal(x)))', step2, step5)
    step7 = prover.add_mp('(Greek(x)->Mortal(x))', step4, step6)
    step8 = prover.add_ug('Ax[(Greek(x)->Mortal(x))]', step7)
    return prover.qed()
```

Steps 5–7 of the above proof seem a bit cumbersome, so in the next task you will write a helper method called `add_tautological_implication()` that provides an easier way to derive a tautological implication of previous lines and allows for the following shorter implementation:

predicates/some_proofs.py
```
def syllogism_all_all_proof_with_tautological_implication(print_as_proof_forms:
                                                          bool = False) -> \
        Proof:
    """Using the method `Prover.add_tautological_implication`, proves from the
    assumptions:

    1. All Greeks are human ('Ax[(Greek(x)->Human(x))]'), and
    2. All humans are mortal ('Ax[(Human(x)->Mortal(x))]')

    the conclusion: All Greeks are mortal ('Ax[(Greek(x)->Mortal(x))]').

    Parameters:
        print_as_proof_forms: flag specifying whether the proof is to be printed
            in real time as it is being constructed.

    Returns:
```

```
        A valid proof, constructed using the method
        `Prover.add_tautological_implication`, of the above inference via
        `AXIOMS`.
    """
    prover = Prover({'Ax[(Greek(x)->Human(x))]', 'Ax[(Human(x)->Mortal(x))]'},
                    print_as_proof_forms)
    step1 = prover.add_assumption('Ax[(Greek(x)->Human(x))]')
    step2 = prover.add_universal_instantiation(
        '(Greek(x)->Human(x))', step1, 'x')
    step3 = prover.add_assumption('Ax[(Human(x)->Mortal(x))]')
    step4 = prover.add_universal_instantiation(
        '(Human(x)->Mortal(x))', step3, 'x')
    step5 = prover.add_tautological_implication(
        '(Greek(x)->Mortal(x))', {step2, step4})
    step6 = prover.add_ug('Ax[(Greek(x)->Mortal(x))]', step5)
    return prover.qed()
```

Similar to the skeleton of a single predicate-logic formula, we define the **propositional skeleton** of a predicate-logic inference as the propositional-logic inference rule obtained from the predicate-logic inference by consistently (across all of the assumptions and the conclusion) replacing each (outermost) subformula whose root is a relation, an equality, or a quantifier, with a new name of a propositional atom. For example, the propositional skeleton of the predicate-logic inference with assumptions '(R(x)|Q(y))' and '~R(x)' and conclusion 'Q(y)' is the propositional-logic inference rule with assumptions '(z1|z2)' and '~z1' and conclusion 'z2'. We call an inference in first-order logic a **(predicate-logic) tautological inference** if its propositional skeleton (a propositional-logic inference rule) is sound.[7]

Task 2. Implement the missing code for the method `add_tautological_implication(implication, line_numbers)`, which adds to the current prover a sequence of validly justified lines, the last of which has the formula `implication`. The derived formula `implication` should be the conclusion of a tautological inference from the lines with the given line numbers.

predicates/prover.py

```
class Prover:
        ⋮
    def add_tautological_implication(self, implication: Union[Formula, str],
                                     line_numbers: AbstractSet[int]) -> int:
        """Appends to the proof being created by the current prover a sequence
        of validly justified lines, the last of which validly justifies the
        given formula, which is the conclusion of a tautological inference whose
        assumptions are the specified already existing lines of the proof.

        Parameters:
            implication: conclusion of the sequence of lines to be appended,
                specified as either a formula or its string representation.
            line_numbers: line numbers in the proof of formulas from which
                conclusion can be a tautologically inferred.
```

[7]We once again use the terminology *the* propositional skeleton somewhat misleadingly, as there are many propositional skeletons for any given predicate-logic inference. For example, the inference rule with assumptions '(z3|z4)' and '~z3' and conclusion 'z4' is also a propositional skeleton of the above predicate-logic inference. There is once again no problem here, though, since either all skeletons of a given predicate-logic inference are sound, or none are.

```
        Returns:
            The line number of the newly appended line that justifies the given
            formula in the proof being created by the current prover.
        """
        if isinstance(implication, str):
            implication = Formula.parse(implication)
        for line_number in line_numbers:
            assert line_number < len(self._lines)
        # Task 10.2
```

Hint: Think back to your solution to Task 4 in Chapter 6 (the implementation of `prove_sound_inference()`), and understand why a given predicate-logic inference is a tautological inference if and only if its "encoding" (predicate-logic) inference is a predicate-logic tautology.

One nice thing about your solution to Task 2 is that it allows you to never bother with "double MP" maneuvers again, even when the original conditional statement (the statement of the form '$(\phi\rightarrow(\psi\rightarrow\xi))$') is not a tautology. If you have somehow proven the three statements '$(\phi\rightarrow(\psi\rightarrow\xi))$', ϕ, and ψ, then instead of deriving ξ from these using a "double MP" maneuver (first deriving '$(\psi\rightarrow\xi)$' via MP, and then deriving ξ via yet another MP), you can instead derive ξ as a tautological implication of these three statements using one Python line.

Let us try to formalize and prove yet another syllogism: All men are mortal, some men exist, and thus some mortals exist.

Assumptions:

1. '∀x[(Man(x)→Mortal(x))]'
2. '∃x[Man(x)]'

Conclusion: '∃x[Mortal(x)]'

Proof:

1. '∀x[(Man(x)→Mortal(x))]'. Justification: first assumption.
2. '∃x[Man(x)]'. Justification: second assumption.
3. '(Man(x)→Mortal(x))'. Justification: universal instantiation of Step 1.
4. '(Mortal(x)→∃x[Mortal(x)])'. Justification: EI with $\phi(\Box)$ defined as 'Mortal(□)' and with x and τ both defined as 'x'.
5. '(Man(x)→∃x[Mortal(x)])'. Justification: tautological inference of Steps 3 and 4.
6. '∀x[(Man(x)→∃x[Mortal(x)])]'. Justification: UG of Step 5.
7. '((∀x[(Man(x)→∃x[Mortal(x)])]&∃x[Man(x)])→∃x[Mortal(x)])'. Justification: ES with $\psi(\Box)$ defined as 'Man(□)', with x defined as 'x', and with ϕ defined as '∃x[Mortal(x)]' (which does not have 'x' free).
8. '∃x[Mortal(x)]'. Justification: tautological inference of Steps 2, 6, and 7.

predicates/some_proofs.py
```
def syllogism_all_exists_proof(print_as_proof_forms: bool = False) -> Proof:
    """Proves from the assumptions:

    1. All men are mortal ('Ax[(Man(x)->Mortal(x))]'), and
    2. Some men exist ('Ex[Man(x)]')

    the conclusion: Some mortals exist ('Ex[Mortal(x)]').

    Parameters:
        print_as_proof_forms: flag specifying whether the proof is to be printed
            in real time as it is being constructed.

    Returns:
        A valid proof of the above inference via
        `~predicates.prover.Prover.AXIOMS`.
    """
    prover = Prover({'Ax[(Man(x)->Mortal(x))]', 'Ex[Man(x)]'},
                    print_as_proof_forms)
    step1 = prover.add_assumption('Ax[(Man(x)->Mortal(x))]')
    step2 = prover.add_assumption('Ex[Man(x)]')
    step3 = prover.add_universal_instantiation(
        '(Man(x)->Mortal(x))', step1, 'x')
    step4 = prover.add_instantiated_assumption(
        '(Mortal(x)->Ex[Mortal(x)])', Prover.EI,
        {'R': 'Mortal(_)', 'c': 'x'})
    step5 = prover.add_tautological_implication(
        '(Man(x)->Ex[Mortal(x)])', {step3, step4})
    step6 = prover.add_ug('Ax[(Man(x)->Ex[Mortal(x)])]', step5)
    step7 = prover.add_instantiated_assumption(
        '((Ax[(Man(x)->Ex[Mortal(x)])]&Ex[Man(x)])->Ex[Mortal(x)])', Prover.ES,
        {'R': 'Man(_)', 'Q': 'Ex[Mortal(x)]'})
    step8 = prover.add_tautological_implication(
        'Ex[Mortal(x)]', {step2, step6, step7})
    return prover.qed()
```

The maneuver in the last three steps of the above proof is quite useful in general, where the idea is that once we have shown that some x with a property '$P(x)$' exists (e.g., `'Ex[Man(x)]'`), and that '$(P(x){\rightarrow}Q)$' (e.g., `'(Man(x)->Ex[Mortal(x)])'`), then we can deduce Q (e.g., `'Ex[Mortal(x)]'`). In the next task you will write a helper method called `add_existential_derivation()` that automates this maneuver and allows for the following shorter implementation:

predicates/some_proofs.py
```
def syllogism_all_exists_proof_with_existential_derivation(print_as_proof_forms:
                                                           bool = False) -> \
        Proof:
    """Using the method `~predicates.prover.Prover.add_existential_derivation`,
    proves from the assumptions:

    1. All men are mortal ('Ax[(Man(x)->Mortal(x))]'), and
    2. Some men exist ('Ex[Man(x)]')

    the conclusion: Some mortals exist ('Ex[Mortal(x)]').

    Parameters:
        print_as_proof_forms: flag specifying whether the proof is to be printed
```

```
            in real time as it is being constructed.

    Returns:
        A valid proof, constructed using the method
        `~predicates.prover.Prover.add_existential_derivation`, of the above
        inference via `~predicates.prover.Prover.AXIOMS`.
    """
    prover = Prover({'Ax[(Man(x)->Mortal(x))]', 'Ex[Man(x)]'},
                    print_as_proof_forms)
    step1 = prover.add_assumption('Ax[(Man(x)->Mortal(x))]')
    step2 = prover.add_assumption('Ex[Man(x)]')
    step3 = prover.add_universal_instantiation(
        '(Man(x)->Mortal(x))', step1, 'x')
    step4 = prover.add_instantiated_assumption(
        '(Mortal(x)->Ex[Mortal(x)])', Prover.EI, {'R': 'Mortal(_)', 'c': 'x'})
    step5 = prover.add_tautological_implication(
        '(Man(x)->Ex[Mortal(x)])', {step3, step4})
    step6 = prover.add_existential_derivation('Ex[Mortal(x)]', step2, step5)
    return prover.qed()
```

Task 3. Implement the missing code for the method `add_existential_derivation(consequent, line_number1, line_number2)`, which adds to the current prover a sequence of validly justified lines, the last of which has the formula `consequent`. The line with number `line_number1` must hold an existential formula '$\exists x[\phi(x)]$' (for some variable x) and the line with number `line_number2` must hold the implication '$(\phi(x)\rightarrow\psi)$', where ψ is the derived formula `consequent`.

predicates/prover.py

```
class Prover:
        ⋮
    def add_existential_derivation(self, consequent: Union[Formula, str],
                                   line_number1: int, line_number2: int) -> int:
        """Appends to the proof being created by the current prover a sequence
        of validly justified lines, the last of which validly justifies the
        given formula, which is the consequent of the second specified already
        existing line of the proof, whose antecedent is existentially quantified
        in the first specified already existing line of the proof.

        Parameters:
            consequent: conclusion of the sequence of lines to be appended,
                specified as either a formula or its string representation.
            line_number1: line number in the proof of an existentially
                quantified formula of the form
                'E`x`[`antecedent(x)`]', where `x` is a variable name that may
                have free occurrences in `antecedent(x)` but has no free
                occurrences in `consequent`.
            line_number2: line number in the proof of the formula
                '(`antecedent(x)`->`consequent`)'.

        Returns:
            The line number of the newly appended line that justifies the given
            formula in the proof being created by the current prover.
        """
        if isinstance(consequent, str):
            consequent = Formula.parse(consequent)
        assert line_number1 < len(self._lines)
        quantified = self._lines[line_number1].formula
```

```
        assert quantified.root == 'E'
        assert quantified.variable not in consequent.free_variables()
        assert line_number2 < len(self._lines)
        conditional = self._lines[line_number2].formula
        assert conditional == Formula('->', quantified.predicate, consequent)
        # Task 10.3
```

It is now finally time for you to prove a few statements on your own.

Task 4. Implement the missing code for the function `lovers_proof()` in the file `predicates/some_proofs.py`, which returns a proof of the following:
Assumptions:

1. Everybody loves somebody: '∀x[∃y[Loves(x,y)]]'.
2. Everybody loves a lover: '∀x[∀z[∀y[(Loves(x,y)→Loves(z,x))]]]'.

Conclusion: Everybody loves everybody: '∀x[∀z[Loves(z,x)]]'.

predicates/some_proofs.py

```
def lovers_proof(print_as_proof_forms: bool = False) -> Proof:
    """Proves from the assumptions:

    1. Everybody loves somebody ('Ax[Ey[Loves(x,y)]]'), and
    2. Everybody loves a lover ('Ax[Az[Ay[(Loves(x,y)->Loves(z,x))]]]')

    the conclusion: Everybody loves everybody ('Ax[Az[Loves(z,x)]]').

    Parameters:
        print_as_proof_forms: flag specifying whether the proof is to be printed
            in real time as it is being constructed.

    Returns:
        A valid proof of the above inference via `AXIOMS`.
    """
    prover = Prover({'Ax[Ey[Loves(x,y)]]',
                     'Ax[Az[Ay[(Loves(x,y)->Loves(z,x))]]]'},
                    print_as_proof_forms)
    # Task 10.4
    return prover.qed()
```

Task 5. Implement the missing code for the function `homework_proof()` in the file `predicates/some_proofs.py`, which returns a proof of the following:
Assumptions:

1. No homework is fun: '~∃x[(Homework(x)&Fun(x))]'.
2. Some homework is reading: '∃x[(Homework(x)&Reading(x))]'.

Conclusion: Some reading is not fun: '∃x[(Reading(x)&~Fun(x))]'.

predicates/some_proofs.py

```
def homework_proof(print_as_proof_forms: bool = False) -> Proof:
    """Proves from the assumptions:

    1. No homework is fun ('~Ex[(Homework(x)&Fun(x))]'), and
    2. Some reading is homework ('Ex[(Homework(x)&Reading(x))]')
```

```
    the conclusion: Some reading is not fun ('Ex[(Reading(x)&~Fun(x))]').

    Parameters:
        print_as_proof_forms: flag specifying whether the proof is to be printed
            in real time as it is being constructed.

    Returns:
        A valid proof of the above inference via `AXIOMS`.
    """
    prover = Prover({'~Ex[(Homework(x)&Fun(x))]',
                     'Ex[(Homework(x)&Reading(x))]'}, print_as_proof_forms)
    # Task 10.5
    return prover.qed()
```

Hint: Notice that for any formula ϕ, we have that 'ϕ→∃x[ϕ]' is an instance of EI. Use this once for deriving '~(Homework(x)&Fun(x))', and once again for deriving '((Reading(x)&~Fun(x))→∃x[(Reading(x)&~Fun(x))])'. Note that since the left-hand side of the latter formula is not true in general, you'll need to use this latter formula in a clever way in your proof in order to derive its right-hand side…but how? Well, since you have not used your second assumption yet, it is time to use it, and since it is existentially quantified, you will have to use it via an existential derivation. Can you see how what you have proven so far can be used to prove the other formula needed for the existential derivation to give you the desired overall conclusion?

3 Some Mathematics

We now move on to using logic to express basic mathematical structures. In particular we will use functions and equality much more.

3.1 Groups

We start with one of the simplest mathematical structures, that of a **group**. While a **field** has two operators: addition and multiplication, a group only has one operator, which we will denote by addition. The **language** in which we will describe a group has, accordingly, two function symbols—a binary function name 'plus' and a *unary* function name 'minus'—and a constant name '0'. A group has only three axioms:
Group Axioms:

1. Zero Axiom: 'plus(0,x)=x'
2. Negation Axiom: 'plus(minus(x),x)=0'
3. Associativity Axiom: 'plus(plus(x,y),z)=plus(x,plus(y,z))'

predicates/some_proofs.py

```
#: The three group axioms
GROUP_AXIOMS = frozenset({'plus(0,x)=x', 'plus(minus(x),x)=0',
                          'plus(plus(x,y),z)=plus(x,plus(y,z))'})
```

While our programs will stick to this simple functional notation, in this chapter we will use the equivalent standard, infix notation for better readability:

1. Zero Axiom: $0 + x = x$
2. Negation Axiom: $-x + x = 0$
3. Associativity Axiom: $(x + y) + z = x + (y + z)$

We note that group addition may possibly be non-commutative, i.e., it is not necessarily the case that $x + y = y + x$. Therefore, since we only defined 0 to be neutral to addition when it is on the left, it is not clear that it is also neutral to addition when it is on the right (i.e., it is not clear that $x + 0 = x$). However, it turns out that one can carefully prove this from the three group axioms, and we will now formulate this proof.
Assumptions: Group Axioms
Conclusion: $x + 0 = x$
Proof:
We will trace and formalize the following mathematical proof, which you may have seen written on the board if you took a course on Algebraic Structures. The basic "trick" of this proof is to add the term $(- - x + -x)$ on the left:

$$x + 0 = 0 + (x + 0) = (0 + x) + 0 = ((- - x + -x) + x) + 0 =$$
$$= (- - x + (-x + x)) + 0 = (- - x + 0) + 0 = - - x + (0 + 0) =$$
$$= - - x + 0 = - - x + (-x + x) = (- - x + -x) + x = 0 + x = x.$$

Let us formalize this proof in our system. We start by listing the axioms as the first steps of the proof:

1. $0 + x = x$. Justification: Zero Axiom.
2. $-x + x = 0$. Justification: Negation Axiom.
3. $(x + y) + z = x + (y + z)$. Justification: Associativity Axiom.

predicates/some_proofs.py

```
def right_neutral_proof(···, print_as_proof_forms: bool = False) -> Proof:
    """Proves from the group axioms that x+0=x ('plus(x,0)=x').

    Parameters:
          ⋮
        print_as_proof_forms: flag specifying whether the proof is to be printed
            in real time as it is being constructed.

    Returns:
        A valid proof of the above inference via `AXIOMS`.
    """
    prover = Prover(GROUP_AXIOMS, print_as_proof_forms)
    zero = prover.add_assumption('plus(0,x)=x')
    negation = prover.add_assumption('plus(minus(x),x)=0')
    associativity = prover.add_assumption('plus(plus(x,y),z)=plus(x,plus(y,z))')
    ⋮
```

We will also want to use the "flipped" equalities that follow from these axioms by the symmetry of equality. While we have not defined the symmetry of equality as a logical axiom, it can be derived from the logical axioms of equality (RX and ME), and the next task will provide a convenient interface to doing so, and performing this kind of flipping.

Task 6. Implement the missing code for the method `add_flipped_equality(flipped, line_number)`, which adds to the current prover a sequence of validly justified lines, the last of which has the formula `flipped`. The derived formula `flipped` must be of the form $\tau = \sigma$ (for some terms τ and σ), where the line with the given `line_number` must hold the "non-flipped" equality $\sigma = \tau$.

predicates/prover.py
```
class Prover:
        ⋮
    def add_flipped_equality(self, flipped: Union[Formula, str],
                             line_number: int) -> int:
        """Appends to the proof being created by the current prover a sequence
        of validly justified lines, the last of which validly justifies the
        given equality, which is the result of exchanging the two sides of an
        equality from the specified already existing line of the proof.

        Parameters:
            flipped: conclusion of the sequence of lines to be appended,
                specified as either a formula or its string representation.
            line_number: line number in the proof of an equality that is the
                same as the given equality, except that the two sides of the
                equality are exchanged.

        Returns:
            The line number of the newly appended line that justifies the given
            equality in the proof being created by the current prover.
        """
        if isinstance(flipped, str):
            flipped = Formula.parse(flipped)
        assert is_equality(flipped.root)
        assert line_number < len(self._lines)
        equality = self._lines[line_number].formula
        assert equality == Formula('=', [flipped.arguments[1],
                                         flipped.arguments[0]])
        # Task 10.6
```

We can continue our proof:

4. $x = 0 + x$. Justification: flipped Zero Axiom.

5. $0 = -x + x$. Justification: flipped Negation Axiom.

6. $x + (y + z) = (x + y) + z$. Justification: flipped Associativity Axiom.

predicates/some_proofs.py
```
def right_neutral_proof(···, print_as_proof_forms: bool = False) -> Proof:
    ⋮
    flipped_zero = prover.add_flipped_equality('x=plus(0,x)', zero)
    flipped_negation = prover.add_flipped_equality(
        '0=plus(minus(x),x)', negation)
    flipped_associativity = prover.add_flipped_equality(
        'plus(x,plus(y,z))=plus(plus(x,y),z)', associativity)
    ⋮
```

Notice that early in the above mathematical proof, we used the equality $0 = --x + -x$, so we should certainly derive it somewhere in our proof. This equation is an instance of the flipped negation axiom, obtained by plugging $-x$ into x. We can derive it in our proof by first applying UG to the flipped negation axiom to obtain $\forall x[0 = -x + x]$, and then using our `add_universal_instantiation()` method, substituting $-x$ into x. The next task will provide a convenient interface to performing this kind of derivation, and will also allow making several substitutions in one call.

Task 7. Implement the missing code for the method `add_free_instantiation(instantiation, line_number, substitution_map)`, which adds to the current prover a sequence of validly justified lines, the last of which has the formula `instantiation`. The derived formula `instantiation` should be the result of substituting free variable names of the formula in the line with the given number with terms, according to the given map, which maps variable names to terms.

predicates/prover.py

```
class Prover:
        ⋮
    def add_free_instantiation(self, instantiation: Union[Formula, str],
                               line_number: int,
                               substitution_map:
                               Mapping[str, Union[Term, str]]) -> int:
        """Appends to the proof being created by the current prover a sequence
        of validly justified lines, the last of which validly justifies the
        given formula, which is the result of substituting terms for the free
        variable names of the formula of the specified already existing line of
        the proof.

        Parameters:
            instantiation: conclusion of the sequence of lines to be appended,
                which contains no variable names starting with ``z``, specified
                as either a formula or its string representation.
            line_number: line number in the proof of a formula with free
                variables, which contains no variable names starting with ``z``.
            substitution_map: mapping from free variable names of the formula
                with the given line number to terms that contain no variable
                names starting with ``z``, to be substituted for them to obtain
                the given formula. Each value of this map may also be given as a
                string representation (instead of a term). Only variable names
                originating in the formula with the given line number are
                substituted (i.e., variable names originating in one of the
                specified substitutions are not subjected to additional
                substitutions).

        Returns:
            The line number of the newly appended line that justifies the given
            formula in the the proof being created by the current prover.

        Examples:
            If Line `line_number` contains the formula
            'Ay[Az[f(x,y)=g(z,y)]]' and `substitution_map` is
            ``{'y': Term.parse('h(w)')}``, then `instantiation` should be
            'Az[f(x,h(w))=g(z,h(w))]'.
        """
        if isinstance(instantiation, str):
            instantiation = Formula.parse(instantiation)
```

```
        assert line_number < len(self._lines)
        substitution_map = dict(substitution_map)
        for variable in substitution_map:
            assert is_variable(variable)
            term = substitution_map[variable]
            if isinstance(term, str):
                substitution_map[variable] = Term.parse(term)
        assert instantiation == \
               self._lines[line_number].formula.substitute(substitution_map)
        # Task 10.7
```

Example: If we have in Line 17 of the proof being constructed by a prover `p` the formula 'plus(x,y)=plus(y,x)', then the call

```
p.add_free_instantiation('plus(f(y),g(x,0))=plus(g(x,0),f(y))', 17,
                         {'x':'f(y)', 'y':'g(x,0)'})
```

will add to the proof a few lines, the last of which proves the formula 'plus(f(y),g(x,0)) =plus(g(x,0),f(y))'.

Guidelines: As mentioned above, substituting a term into a single variable is easy by calling the method `add_ug()` and then calling the method `add_universal_instantiation()`. While in simple cases this could be done sequentially for all variables in the given substitution map, a sequential substitution will not give the required results if the substituted terms themselves contain in them some of the substituted variables. For instance, in the example above, if we first substituted 'f(y)' into 'x' to obtain the intermediate formula 'plus(f(y),y)=plus(y,f(y))', then a second-stage replacement of 'y' with 'g(x,0)' would also *incorrectly* cause 'f(y)' to be replaced with 'f(g(x,0))', obtaining the formula 'plus(f(g(x,0)),g(x,0))=plus(g(x,0),f(g(x,0)))' instead of the requested 'plus(f(y),g(x,0)) =plus(g(x,0),f(y))'. To avoid this, first sequentially replace all variables that need to be instantiated with new variable names, e.g., in our example obtaining an intermediate formula 'plus(z1,z2)=plus(z2,z1)' (remember that `next(fresh_variable_name_generator)` is your friend…), and then instantiate each of these temporary unique variables with the target term.

We can now obtain arbitrary instances of the basic rules, so here is a good place in our proof to list those that we will need:

7. $0 = - - x + -x$. Justification: free instantiation of the flipped Negation Axiom, substituting x with $-x$.

8. $- - x + -x = 0$. Justification: flipped equality of Step 7.

9. $(- - x + -x) + x = - - x + (-x + x)$. Justification: free instantiation of the Associativity Axiom, substituting x with $- - x$, substituting y with $-x$, and substituting z with x.

10. $0+0 = 0$. Justification: free instantiation of the Zero Axiom, substituting x with 0.

predicates/some_proofs.py

```
def right_neutral_proof(···, print_as_proof_forms: bool = False) -> Proof:
    ⋮
    step7 = prover.add_free_instantiation(
        '0=plus(minus(minus(x)),minus(x))', flipped_negation, {'x': 'minus(x)'})
```

```
    step8 = prover.add_flipped_equality(
        'plus(minus(minus(x)),minus(x))=0', step7)
    step9 = prover.add_free_instantiation(
        'plus(plus(minus(minus(x)),minus(x)),x)='
        'plus(minus(minus(x)),plus(minus(x),x))',
        associativity, {'x': 'minus(minus(x))', 'y': 'minus(x)', 'z': 'x'})
    step10 = prover.add_free_instantiation('plus(0,0)=0', zero, {'x': '0'})
    ⋮
```

We can now "really start" tracing the mathematical proof above, step by step:

11. $x + 0 = 0 + (x + 0)$. Justification: free instantiation of the flipped Zero Axiom, substituting x with $(x + 0)$.

12. $0+(x+0) = (0+x)+0$. Justification: free instantiation of the flipped Associativity Axiom, substituting x and z with 0, and substituting y with x.

predicates/some_proofs.py

```
def right_neutral_proof(···, print_as_proof_forms: bool = False) -> Proof:
    ⋮
    step11 = prover.add_free_instantiation(
        'plus(x,0)=plus(0,plus(x,0))', flipped_zero, {'x': 'plus(x,0)'})
    step12 = prover.add_free_instantiation(
        'plus(0,plus(x,0))=plus(plus(0,x),0)', flipped_associativity,
        {'x': '0', 'y': 'x', 'z': '0'})
    ⋮
```

The next thing that we would like to deduce is $(0+x)+0 = ((--x+-x)+x)+0$, by "substituting" both sides of the equality $0 = --x+-x$ from Step 7 into the expression $(\square + x) + 0$. This type of substitution can be performed using the logical axioms of equality (RX and ME), and your solution to the following task will provide a convenient interface to performing it.

Task 8. Implement the missing code for the method `add_substituted_equality(substituted, line_number, parametrized_term)`, which adds to the current prover a sequence of validly justified lines, the last of which has the formula `substituted`. The line with number `line_number` must hold an equality 'τ=σ' (for some terms τ and σ) and the derived formula `substituted` should be '$\phi(\tau)$=$\phi(\sigma)$', where $\phi(\square)$ is the given parametrized term.

predicates/prover.py

```
class Prover:
        ⋮
    def add_substituted_equality(self, substituted: Union[Formula, str],
                                 line_number: int,
                                 parametrized_term: Union[Term, str]) -> int:
        """Appends to the proof being created by the current prover a sequence
        of validly justified lines, the last of which validly justifies the
        given equality, whose two sides are the results of substituting the
        two respective sides of an equality from the specified already existing
        line of the proof into the given parametrized term.

        Parameters:
```

```
        substituted: conclusion of the sequence of lines to be appended,
            specified as either a formula or its string representation.
        line_number: line number in the proof of an equality.
        parametrized_term: term parametrized by the constant name '_',
            specified as either a term or its string representation, such
            that substituting each of the two sides of the equality with the
            given line number into this parametrized term respectively
            yields each of the two sides of the given equality.

    Returns:
        The line number of the newly appended line that justifies the given
        equality in the proof being created by the current prover.

    Examples:
        If Line `line_number` contains the formula 'g(x)=h(y)' and
        `parametrized_term` is '_+7', then `substituted` should be
        'g(x)+7=h(y)+7'.
    """
    if isinstance(substituted, str):
        substituted = Formula.parse(substituted)
    assert is_equality(substituted.root)
    assert line_number < len(self._lines)
    equality = self._lines[line_number].formula
    assert is_equality(equality.root)
    if isinstance(parametrized_term, str):
        parametrized_term = Term.parse(parametrized_term)
    assert substituted == \
           Formula('=', [parametrized_term.substitute(
                              {'_': equality.arguments[0]}),
                         parametrized_term.substitute(
                              {'_': equality.arguments[1]})])
    # Task 10.8
```

We can now continue with our proof:

13. $(0 + x) + 0 = ((- - x + -x) + x) + 0$. Justification: substituting both sides of the equality $0 = - - x + -x$ from Step 7 into the expression $(\Box + x) + 0$.

14. $((- - x + -x) + x) + 0 = (- - x + (-x + x)) + 0$. Justification: substituting both sides of the equality $(- - x + -x) + x = - - x + (-x + x)$ from Step 9 into the expression $\Box + 0$.

15. $(- - x + (-x + x)) + 0 = (- - x + 0) + 0$. Justification: substituting both sides of the equality $-x + x = 0$ from Step 2 into the expression $(- - x + \Box) + 0$.

16. $(- - x + 0) + 0 = - - x + (0 + 0)$. Justification: free instantiation of the Associativity Axiom, substituting x with $- - x$ and substituting y and z with 0.

17. $- - x + (0 + 0) = - - x + 0$. Justification: substituting both sides of the equality $0 + 0 = 0$ from Step 10 into the expression $- - x + \Box$.

18. $- - x + 0 = - - x + (-x + x)$. Justification: substituting both sides of the equality $0 = -x + x$ from Step 5 into the expression $- - x + \Box$.

19. $--x+(-x+x)=(--x+-x)+x$. Justification: free instantiation of the flipped Associativity Axiom, substituting x with $--x$, substituting y with $-x$, and substituting z with x.

20. $(--x+-x)+x=0+x$. Justification: substituting in both sides of the equality $--x+-x=0$ from Step 8 into the expression $\Box+x$.

predicates/some_proofs.py

```
def right_neutral_proof(···, print_as_proof_forms: bool = False) -> Proof:
    ⋮
    step13 = prover.add_substituted_equality(
        'plus(plus(0,x),0)=plus(plus(plus(minus(minus(x)),minus(x)),x),0)',
        step7, 'plus(plus(_,x),0)')
    step14 = prover.add_substituted_equality(
        'plus(plus(plus(minus(minus(x)),minus(x)),x),0)='
        'plus(plus(minus(minus(x)),plus(minus(x),x)),0)',
        step9, 'plus(_,0)')
    step15 = prover.add_substituted_equality(
        'plus(plus(minus(minus(x)),plus(minus(x),x)),0)='
        'plus(plus(minus(minus(x)),0),0)',
        negation, 'plus(plus(minus(minus(x)),_),0)')
    step16 = prover.add_free_instantiation(
        'plus(plus(minus(minus(x)),0),0)=plus(minus(minus(x)),plus(0,0))',
        associativity, {'x': 'minus(minus(x))', 'y': '0', 'z': '0'})
    step17 = prover.add_substituted_equality(
        'plus(minus(minus(x)),plus(0,0))=plus(minus(minus(x)),0)',
        step10, 'plus(minus(minus(x)),_)')
    step18 = prover.add_substituted_equality(
        'plus(minus(minus(x)),0)=plus(minus(minus(x)),plus(minus(x),x))',
        flipped_negation, 'plus(minus(minus(x)),_)')
    step19 = prover.add_free_instantiation(
        'plus(minus(minus(x)),plus(minus(x),x))='
        'plus(plus(minus(minus(x)),minus(x)),x)', flipped_associativity,
        {'x': 'minus(minus(x))','y': 'minus(x)','z': 'x'})
    step20 = prover.add_substituted_equality(
        'plus(plus(minus(minus(x)),minus(x)),x)=plus(0,x)', step8, 'plus(_,x)')
    ⋮
```

Recalling that the Zero Axiom gives $0+x=x$, we now have a sequence of equalities (Steps 11–20 in order, followed by Step 1) that we would like to "chain" together using the transitivity of equality to get the final conclusion that we are after. While we have not defined the transitivity of equality as a logical axiom, it can be derived from ME, and your solution to the next task will provide a convenient interface to doing so, and performing this kind of chaining.

Task 9. Implement the missing code for the method `add_chained_equality(chained, line_numbers)`, which adds to the current prover a sequence of validly justified lines, the last of which has the formula `chained`. The derived formula `chained` must be of the form 'τ=σ', where the lines with the given `line_numbers` hold (in the given order) a sequence of equalities 'τ_1=τ_2', 'τ_2=τ_3', ..., 'τ_{k-1}=τ_k', with τ_1 being τ and τ_k being σ.

predicates/prover.py

```
class Prover:
        ⋮
    def _add_chaining_of_two_equalities(self, line_number1: int,
```

```
                                line_number2: int) -> int:
        """Appends to the proof being created by the current prover a sequence
        of validly justified lines, the last of which validly justifies an
        equality that is the result of chaining together two equalities from
        the specified already existing lines of the proof.

        Parameters:
            line_number1: line number in the proof of an equality of the form
                '`first`=`second`'.
            line_number2: line number in the proof of an equality of the form
                '`second`=`third`'.

        Returns:
            The line number of the newly appended line that justifies the
            equality '`first`=`third`' in the proof being created by the current
            prover.

        Examples:
            If Line `line_number1` contains the formula 'a=b' and Line
            `line_number2` contains the formula 'b=f(b)', then the last appended
            line will contain the formula 'a=f(b)'.
        """
        assert line_number1 < len(self._lines)
        equality1 = self._lines[line_number1].formula
        assert is_equality(equality1.root)
        assert line_number2 < len(self._lines)
        equality2 = self._lines[line_number2].formula
        assert is_equality(equality2.root)
        assert equality1.arguments[1] == equality2.arguments[0]
        # Task 10.9.1

    def add_chained_equality(self, chained: Union[Formula, str],
                             line_numbers: Sequence[int]) -> int:
        """Appends to the proof being created by the current prover a sequence
        of validly justified lines, the last of which validly justifies the
        given equality, which is the result of chaining together equalities
        from the specified already existing lines of the proof.

        Parameters:
            chained: conclusion of the sequence of lines to be appended,
                specified as either a formula or its string representation,
                of the form '`first`=`last`'.
            line_numbers: line numbers in the proof of equalities of the form
                '`first`=`second`', '`first`=`third`', ...,
                '`before_last`=`last`', i.e., the left-hand side of the first
                equality is the left-hand side of the given equality, the
                right-hand of each equality (except for the last) is the
                left-hand side of the next equality, and the right-hand side of
                the last equality is the right-hand side of the given equality.

        Returns:
            The line number of the newly appended line that justifies the given
            equality in the proof being created by the current prover.

            Examples:
            If `line_numbers` is ``[7,3,9]``, Line 7 contains the formula
            'a=b', Line 3 contains the formula 'b=f(b)', and Line 9 contains the
             formula 'f(b)=0', then `chained` should be 'a=0'.
```

```
        """
        if isinstance(chained, str):
            chained = Formula.parse(chained)
        assert is_equality(chained.root)
        assert len(line_numbers) >= 2
        current_term = chained.arguments[0]
        for line_number in line_numbers:
            assert line_number < len(self._lines)
            equality = self._lines[line_number].formula
            assert is_equality(equality.root)
            assert equality.arguments[0] == current_term
            current_term = equality.arguments[1]
        assert chained.arguments[1] == current_term
        # Task 10.9.2
```

Guidelines: First implement the missing code for the private static method `_add_chaining_of_two_equalities(line_number1, line_number2)` that has similar functionality but with only two equalities to chain (see the method *docstring* for details), and then use that method to solve this task in full generality.

So can now finally conclude our proof:

21. $x + 0 = x$. Justification: chaining Steps 11–20 in order, followed by Step 1.

predicates/some_proofs.py

```
def right_neutral_proof(···, print_as_proof_forms: bool = False) -> Proof:
    ⋮
    step21 = prover.add_chained_equality(
        'plus(x,0)=x',
        [step11, step12, step13, step14, step15, step16, step17, step18, step19,
         step20, zero])
    return prover.qed()
```

During this proof we have developed enough helper functions to prepare us for the remainder of this chapter, in which you will prove additional important mathematical theorems. We start by showing that not only is zero neutral to addition both on the left and on the right, but this property is also unique to zero.

Task 10. Implement the missing code for the function `unique_zero_proof()` in the file `predicates/some_proofs.py`, which returns a proof of the following:
Assumptions: Group Axioms and $a + c = a$.
Conclusion: $c = 0$.

predicates/some_proofs.py

```
def unique_zero_proof(print_as_proof_forms: bool = False) -> Proof:
    """Proves from the group axioms and from the assumption a+c=a
    ('plus(a,c)=a') that c=0 ('c=0').

    Parameters:
        print_as_proof_forms: flag specifying whether the proof is to be printed
            in real time as it is being constructed.

    Returns:
        A valid proof of the above inference via `AXIOMS`.
    """
    prover = Prover(GROUP_AXIOMS.union({'plus(a,c)=a'}), print_as_proof_forms)
    # Task 10.10
    return prover.qed()
```

3.2 Fields

We move on from groups to fields. We will represent addition in a **field** using the function name 'plus', multiplication in a field using the function name 'times', zero (the neutral to additivity) using the constant name '0', and one (the neutral to multiplication) using the constant name '1'.

Field Axioms:

1. $0 + x = x$
2. $-x + x = 0$
3. $(x + y) + z = x + (y + z)$
4. $x + y = y + x$
5. $x \cdot 1 = x$
6. $x \cdot y = y \cdot x$
7. $(x \cdot y) \cdot z = x \cdot (y \cdot z)$
8. $(x \neq 0 \rightarrow \exists y[y \cdot x = 1])$ (where $x \neq 0$ should be read as '~x=0', of course)
9. $x \cdot (y + z) = (x \cdot y) + (x \cdot z)$

predicates/some_proofs.py

```
#: The six field axioms
FIELD_AXIOMS = frozenset(GROUP_AXIOMS.union(
    {'plus(x,y)=plus(y,x)', 'times(x,1)=x', 'times(x,y)=times(y,x)',
     'times(times(x,y),z)=times(x,times(y,z))', '(~x=0->Ey[times(y,x)=1])',
     'times(x,plus(y,z))=plus(times(x,y),times(x,z))'}))
```

Task 11. Implement the missing code for the function `multiply_zero_proof()` in the file `predicates/some_proofs.py`, which returns a proof of the following:
Assumptions: Field Axioms.
Conclusion: $0 \cdot x = 0$.

predicates/some_proofs.py

```
def multiply_zero_proof(print_as_proof_forms: bool = False) -> Proof:
    """Proves from the field axioms that 0*x=0 ('times(0,x)=0').

    Parameters:
        print_as_proof_forms: flag specifying whether the proof is to be printed
            in real time as it is being constructed.

    Returns:
        A valid proof of the above inference via `AXIOMS`.
    """
    prover = Prover(FIELD_AXIOMS, print_as_proof_forms)
    # Task 10.11
    return prover.qed()
```

Hint: If you have seen a proof of this in a Linear Algebra course, you can try to formalize that proof. Alternatively, one possible proof strategy is to first prove that $0 \cdot x + 0 \cdot x = 0 \cdot x$, and to then continue similarly to Task 10. (Note that the field axioms for addition contain all of the group axioms. However, we do not have a convenient interface for inlining the solution to Task 10 in another proof as is, because of its assumption, so feel free to duplicate code rather than build an inlining interface just for the sake of this task.)

3.3 Peano Arithmetic

Peano arithmetic, named after the 19th-century Italian mathematician Giuseppe Peano, model the natural numbers. (In Logic courses and Set Theory courses, the natural numbers customarily start from zero rather than from one.) In Peano arithmetic, apart from multiplication and addition, we have the unary "successor" function, name 's', which should be thought of as returning one when applied to zero, two when applied to one, etc.

Peano Axioms:

1. $(s(x) = s(y) \rightarrow x = y)$
2. $(x \neq 0 \rightarrow \exists y[s(y) = x])$
3. $s(x) \neq 0$
4. $x + 0 = x$
5. $x + s(y) = s(x + y)$
6. $x \cdot 0 = 0$
7. $x \cdot s(y) = x \cdot y + x$
8. Axiom of Induction: the schema $((\phi(0)\&\forall x[(\phi(x) \rightarrow \phi(s(x)))]) \rightarrow \forall x[\phi(x)])$, where $\phi(\Box)$ is a placeholder for a parametrized formula.

predicates/some_proofs.py

```
#: The induction axiom
INDUCTION_AXIOM = Schema(
    Formula.parse('((R(0)&Ax[(R(x)->R(s(x)))])->Ax[R(x)])'), {'R'})
#: The six axioms of Peano arithmetic
PEANO_AXIOMS = frozenset({'(s(x)=s(y)->x=y)', '(~x=0->Ey[s(y)=x])', '~s(x)=0',
                          'plus(x,0)=x', 'plus(x,s(y))=s(plus(x,y))',
                          'times(x,0)=0', 'times(x,s(y))=plus(times(x,y),x)',
                          INDUCTION_AXIOM})
```

Note that we do not have commutativity as an axiom of Peano arithmetic, and it needs to be proven from the given ones. The first step toward proving this is to prove that $0 + x = x$ (note that we only have $x + 0 = x$ is an axiom), which you are now asked to do.

Task 12. Implement the missing code for the function `peano_zero_proof()` in the file `predicates/some_proofs.py`, which returns a proof of the following:
Assumptions: Peano Axioms.
Conclusion: $0 + x = x$.

predicates/some_proofs.py

```
def peano_zero_proof(print_as_proof_forms: bool = False) -> Proof:
    """Proves from the axioms of Peano arithmetic that 0+x=x ('plus(0,x)=x').

    Parameters:
        print_as_proof_forms: flag specifying whether the proof is to be printed
            in real time as it is being constructed.

    Returns:
```

```
        A valid proof of the above inference via
        `~predicates.prover.Prover.AXIOMS`.
    """
    prover = Prover(PEANO_AXIOMS, print_as_proof_forms)
    # Task 10.12
    return prover.qed()
```

Hint: Use induction on x (i.e., use the Axiom of Induction schema with $0 + \Box = \Box$ as $\phi(\Box)$). Whenever you get stuck while trying to prove the induction step, try to use ME.

3.4 Zermelo–Fraenkel Set Theory

Predicate logic turns out to suffice for capturing all of Mathematics from a handful of axioms. The usual formalization is to have **sets** as the basic building blocks (elements) of our universe, and to define everything from there. That is, in terms of first-order logic, there is a single binary relation that denotes membership of an item in a set, 'In(x,y)', meaning $x \in y$.

The axioms for sets are stated so that they imply that an empty set exists (we may or may not have a constant name $\emptyset$ that denotes it), and once we have the empty set we can continue to define the natural numbers as: $0 = \emptyset$, $1 = \{\emptyset\}$, $2 = \{\emptyset, \{\emptyset\}\}$, $3 = \{\emptyset, \{\emptyset\}, \{\emptyset, \{\emptyset\}\}\}$, etc.,[8] and the Set Theory axioms will suffice for proving all the required properties of natural numbers (in particular all the axioms of Peano arithmetic). Notice however that this construction also provides *sets* of natural numbers, and sets of sets of natural numbers, etc., which Peano arithmetic does not provide. Once we have natural numbers, we can continue defining integers, rationals, real numbers, complex numbers, real-valued functions, vector fields, and the rest of Mathematics.

Some of the Set Theory axioms give basic intended properties for sets. For example, the Extensionality Axiom states that two sets are equal (are the same set) if they have the same elements. Formally, '∀x[∀y[(∀z[((In(z,x)→In(z,y))&(In(z,y)→In(z,x)))]→x=y)]]'. Most of the axioms, however, are devoted to ensuring that certain types of sets exist.

To understand the need for this, let us look at our naïve notion of how we get a set: we usually specify a condition and then look at the set of all elements that satisfy it. Formally, for every parametrized formula (condition) $\phi(\Box)$ we imagine an axiom schema of "comprehension" stating that the set of all elements satisfying this condition exists: $\exists y[\forall x[x \in y \leftrightarrow \phi(x)]]$.

predicates/some_proofs.py

```
#: The axiom schema of (unrestricted) comprehension
COMPREHENSION_AXIOM = Schema(
    Formula.parse('Ey[Ax[((In(x,y)->R(x))&(R(x)->In(x,y)))]]'), {'R'})
```

However, in 1901, the famous British philosopher, logician, mathematician, historian, writer, and Nobel laureate (in Literature!) Bertrand Russell, noticed what has come to be known as "Russell's Paradox": that by looking at the set $\{x | x \notin x\}$, the Axiom Schema of Comprehension turns out to lead to a contradiction. You will now formalize his argument.

Task 13. Implement the missing code for the function `russell_paradox_proof()` in the file `predicates/some_proofs.py`, which returns a proof of the following:

[8] For more details, you are highly encouraged to take a course on Set Theory.

Assumptions: Axiom Schema of Comprehension.
Conclusion: The contradiction '(z=z & z≠z)'.

predicates/some_proofs.py
```
def russell_paradox_proof(print_as_proof_forms: bool = False) -> Proof:
    """Proves from the axioms schema of unrestricted comprehension the
    contradiction '(z=z&~z=z)'.

    Parameters:
        print_as_proof_forms: flag specifying whether the proof is to be printed
            in real time as it is being constructed.

    Returns:
        A valid proof of the above inference via
        `~predicates.prover.Prover.AXIOMS`.
    """
    prover = Prover({COMPREHENSION_AXIOM}, print_as_proof_forms)
    # Task 10.13
    return prover.qed()
```

Hint: Following Russell's Paradox, instantiate the Axiom Schema of Unrestricted Comprehension by defining '$\phi(\square)$' as '~In(□,□)'.

We conclude that we cannot just assume that there is a set for any condition that we wish to use (like we would have wanted the Axiom Schema of Comprehension to guarantee), but rather need axioms to tell us which sets exist. In particular, instead of the general Axiom Schema of Comprehension, it is customary to have a weaker Axiom Schema of Specification that only allows imposing conditions on elements of a given set: for every parametrized formula (condition) $\phi(\square)$ the following is an axiom: $\forall z[\exists y[\forall x[x \in y \leftrightarrow (x \in z \,\&\, \phi(x))]]]$. This allows one to take arbitrary subsets of a give set. A number of other Set Theory axioms specify the ways in which one may go "up," that is, build larger sets from those one already has: by taking unions, by taking a power set, by pairing two items, by taking a functional image of a set, and there is also an axiom that guarantees the existence of an infinite set. Beyond these, there is an Axiom of Foundation that essentially states that there cannot be "cycles" of inclusion such as $x \in y \in z \in x$. All of these axioms together are the axiomatic basis for the celebrated Zermelo–Fraenkel (ZF) Set Theory.

Finally, mathematicians also assume, when needed, the Axiom of Choice that states that for every set Z of non-empty sets there exists a function f that chooses, for each $Y \in Z$, some element x in it. The resulting axiomatic system, called ZFC (Zermelo–Fraenkel with Choice) forms the canonical basis of modern Mathematics.