Chapter 7:

Predicate Logic Syntax and Semantics

Propositional Logic, which we studied in the first part of this book up to this point, is not rich enough by itself to represent many common logical statements. Consider for example the simple **syllogism**: All men are mortal, some men exist; thus, some mortals exist. Propositional Logic cannot even express the notions of "all" or "exists," let alone this type of logical deduction. We will therefore now switch to a different, richer logic, called **First-Order Predicate Logic**, or for short, **Predicate Logic** (or **First-Order Logic**). This logic, which will be our language for the second part of this book, will be strong enough to express and formalize such notions.

For example the above statement can be written in Predicate Logic as:[1]

Assumptions: '∀x[(Man(x)→Mortal(x))]', '∃x[Man(x)]'
Conclusion: '∃x[Mortal(x)]'

In fact, this logic is strong enough to represent every statement (and proof) that you have ever seen in Mathematics! For example, the commutative law of addition can be represented as '∀x[∀y[x+y=y+x]]', which we will actually write in the slightly more cumbersome notation '∀x[∀y[plus(x,y)=plus(y,x)]]'. An integral part of allowing quantifications ("for all" / "exists") is that the reasoning is about variables that are no longer only placeholders for Boolean values, but rather placeholders for much richer "values." For example, wherever it appears in the syllogism example above, the variable 'x' can be defined as a placeholder for any living thing, while in the commutative law above, the variables 'x' and 'y' can be placeholders for any number, or more generally for any group element or field element.

This new logic will enable a transformation in our thinking: until this point the formal logic that we were analyzing, Propositional Logic, was different—weaker—than the mathematical language (or programming) with which we were analyzing it. Now, since Predicate Logic is in fact a proper formalization of *all* of Mathematics, the mathematical language that we will use to talk about logic can in fact itself be formalized as Predicate Logic. We will nonetheless keep allowing ourselves to talk "like in normal Math courses" (as you have done since the beginning of your studies) rather than being 100% formal, but it is important to keep in mind that in principle we could convert all of the definitions and proofs in this whole book—or in any other Mathematics book—to a completely formal form written entirely in Predicate Logic. This chapter, which parallels Chapters 1 and 2 only for Predicate Logic, defines the syntax and semantics of Predicate Logic.

[1]Notice that it takes some ingenuity to formalize "All men are mortal" as '∀x[(Man(x)→Mortal(x))]'. Indeed, formalizing human-language sentences as formulas may at times be far from a trivial task, but that is not the focus of this book.

1 Syntax

Propositional Logic was created to reason about Boolean objects; therefore, every formula represents (that is, when we endow it with semantics) a Boolean statement. As we have noted above, in Predicate Logic we will have **formulas** that represent a Boolean statement, such as 'plus(x,y)=z', but we will also have more basic "formula-like" expressions called **terms**, such as 'plus(x,y)', which evaluate to not-necessarily-Boolean values such as numbers, other mathematical objects, or even people or more general living beings as in the case of the term 'x' in the example at the top of this chapter. As with Propositional Logic, we start by syntactically defining terms and formulas in Predicate Logic, and will only later give them semantic meaning. Nonetheless, remembering that formulas will eventually represent Boolean statements, while terms will eventually evaluate to arbitrary not-necessarily-Boolean values, will add an intuitive layer of understanding to our syntactic definitions and will help us understand where we are headed. We start by defining terms in Predicate Logic:

Definition (Term)**.** The following strings are (valid[2]) **terms** in Predicate Logic:

- A **variable name**: a sequence of alphanumeric characters that begins with a letter in 'u'...'z'. For example, 'x', 'y12', or 'zLast'.
- A **constant name**: a sequence of alphanumeric characters that begins with a digit or with a letter in 'a'...'e'; or an underscore (with nothing before or after it). For example, '0', 'c1', '7x', or '_'.
- An n-ary **function invocation** of the form '$f(t_1,\ldots,t_n)$', where f is a **function name** denoted by a sequence of alphanumeric characters that begins with a letter in 'f'...'t', where[3] $n \geq 1$, and where each t_i is itself a (valid) term. For example, 'plus(x,y)', 's(s(0))', or 'f(g(x),h(7,y),c)'.

These are the only (valid) terms in Predicate Logic.

The file `predicates/syntax.py` defines (among other classes) the Python class `Term` for holding a term as a data structure.

predicates/syntax.py

```
def is_constant(string: str) -> bool:
    """Checks if the given string is a constant name.

    Parameters:
        string: string to check.

    Returns:
        ``True`` if the given string is a constant name, ``False`` otherwise.
    """
    return  (((string[0] >= '0' and string[0] <= '9') or \
              (string[0] >= 'a' and string[0] <= 'e')) and \
             string.isalnum()) or string == '_'
```

[2]Similarly to the case of propositional formulas, what we call **valid** terms are often called **well-formed** terms in other textbooks.

[3]Another choice that we could have made would have been to not allow any constants, but to instead allow nullary functions, which would have "acted as" constants. The reason that we specifically allow constants (and therefore disallow nullary functions as they are not needed when constants are allowed), beyond avoiding the clutter of many empty pairs of brackets, will become clear in Chapter 12.

```
def is_variable(string: str) -> bool:
    """Checks if the given string is a variable name.

    Parameters:
        string: string to check.

    Returns:
        ``True`` if the given string is a variable name, ``False`` otherwise.
    """
    return string[0] >= 'u' and string[0] <= 'z' and string.isalnum()

def is_function(string: str) -> bool:
    """Checks if the given string is a function name.

    Parameters:
        string: string to check.

    Returns:
        ``True`` if the given string is a function name, ``False`` otherwise.
    """
    return string[0] >= 'f' and string[0] <= 't' and string.isalnum()

@frozen
class Term:
    """An immutable predicate-logic term in tree representation, composed from
    variable names and constant names, and function names applied to them.

    Attributes:
        root: the constant name, variable name, or function name at the root of
            the term tree.
        arguments: the arguments of the root, if the root is a function name.
    """
    root: str
    arguments: Optional[Tuple[Term, ...]]

    def __init__(self, root: str, arguments: Optional[Sequence[Term]] = None):
        """Initializes a `Term` from its root and root arguments.

        Parameters:
            root: the root for the formula tree.
            arguments: the arguments for the root, if the root is a function
                name.
        """
        if is_constant(root) or is_variable(root):
            assert arguments is None
            self.root = root
        else:
            assert is_function(root)
            assert arguments is not None and len(arguments) > 0
            self.root = root
            self.arguments = tuple(arguments)
```

The constructor of this class creates an expression-tree representations for a term. For example, the data structure for representing the term 'plus(s(x),3))' is constructed using the following code:

```
my_term = Term('plus', [Term('s', [Term('x')]), Term('3')])
```

Note that the terms that serve as arguments of function invocations are passed to the

constructor together as a Python `list` rather than each argument separately (as was the case for, e.g., passing operands to operators in our code for Propositional Logic). We now move on to use terms to define formulas in Predicate Logic:

Definition (Formula)**.** The following strings are (valid[4]) **formulas** in Predicate Logic:

- An **equality** of the form 't_1=t_2', where each of t_1 and t_2 is a (valid) term. For example, '0=0', 's(0)=1', or 'plus(x,y)=plus(y,x)'.
- An n-ary **relation invocation**[5] of the form '$R(t_1,\ldots,t_n)$', where R is a **relation name** denoted by a string of alphanumeric characters that begins with a letter in 'F'...'T', where $n \geq 0$ (note that we specifically allow nullary relations), and where each t_i is a term. For example, 'R(x,y)', 'Plus(s(0),x,s(x))', or 'Q()'.
- A (unary) **negation** of the form '~ϕ', where ϕ is a (valid) formula.
- A **binary operation** of the form '(ϕ*ψ)', where * is one of the binary operators '|', '&', or '$\rightarrow$',[6] and each of ϕ and ψ is a formula.
- A **quantification** of the form '$Qx[\phi]$', where Q is either the **universal quantifier** '$\forall$' which we represent in Python as `'A'` or the **existential quantifier** '$\exists$' which we represent in Python as `'E'`, where x is a variable name (denoted by a sequence of alphanumeric characters that begins a letter in 'u'...'z', as defined above), and where ϕ is a formula. The subformula ϕ that is within the square brackets in the quantification '$Qx[\phi]$' is called the **scope** of the quantification. For example, '$\forall$x[x=x]', '$\exists$x[R(7,y)]', '$\forall$x[$\exists$y[R(x,y)]]', or '$\forall$x[(R(x)|$\exists$x[Q(x)])]'.

These are the only (valid) formulas in Predicate Logic.

The file `predicates/syntax.py` also defines the Python class `Formula` for holding a predicate-logic formula as a data structure.

predicates/syntax.py

```
def is_equality(string: str) -> bool:
    """Checks if the given string is the equality relation.

    Parameters:
        string: string to check.

    Returns:
        ``True`` if the given string is the equality relation, ``False``
        otherwise.
```

[4]As in propositional logic, what we call **valid** formulas are often called **well-formed** formulas in other textbooks.

[5]A relation invocation is sometimes called a **predication** in other textbooks. We use the term relation invocation to stress its technical similarity to a function invocation: both function names and relation names can be invoked on an n-tuple of terms, with the difference being that a function invocation is a term while a relation invocation is a formula. (When we endow them with semantics later, a relation will correspond to a mapping that returns a Boolean value—a truth value—while a function will correspond to a mapping that returns a not-necessarily-Boolean object of the same type, in some sense, as its inputs.)

[6]We could have again used any other (larger or smaller) set of complete Boolean operators, including operators of various arities, as discussed in Chapter 3 for Propositional Logic. As the discussion for Predicate Logic would have been completely equivalent, we omit it here and stick with the unary negation operator and with these three binary operators for convenience as we did in the first part of this book. (We allow ourselves not to bother with nullary operators, though, to avoid terminologically confusing them with the similarly named predicate-logic constants.)

```
    """
    return string == '='

def is_relation(string: str) -> bool:
    """Checks if the given string is a relation name.

    Parameters:
        string: string to check.

    Returns:
        ``True`` if the given string is a relation name, ``False`` otherwise.
    """
    return string[0] >= 'F' and string[0] <= 'T' and string.isalnum()

def is_unary(string: str) -> bool:
    """Checks if the given string is a unary operator.

    Parameters:
        string: string to check.

    Returns:
        ``True`` if the given string is a unary operator, ``False`` otherwise.
    """
    return string == '~'

def is_binary(string: str) -> bool:
    """Checks if the given string is a binary operator.

    Parameters:
        string: string to check.

    Returns:
        ``True`` if the given string is a binary operator, ``False`` otherwise.
    """
    return string == '&' or string == '|' or string == '->'

def is_quantifier(string: str) -> bool:
    """Checks if the given string is a quantifier.

    Parameters:
        string: string to check.

    Returns:
        ``True`` if the given string is a quantifier, ``False`` otherwise.
    """
    return string == 'A' or string == 'E'

@frozen
class Formula:
    """An immutable predicate-logic formula in tree representation, composed
    from relation names applied to predicate-logic terms, and operators and
    quantifications applied to them.

    Attributes:
        root: the relation name, equality relation, operator, or quantifier at
            the root of the formula tree.
        arguments: the arguments of the root, if the root is a relation name or
            the equality relation.
```

```
        first: the first operand of the root, if the root is a unary or binary
            operator.
        second: the second operand of the root, if the root is a binary
            operator.
        variable: the variable name quantified by the root, if the root is a
            quantification.
        statement: the statement quantified by the root, if the root is a
            quantification.
    """
    root: str
    arguments: Optional[Tuple[Term, ...]]
    first: Optional[Formula]
    second: Optional[Formula]
    variable: Optional[str]
    statement: Optional[Formula]

    def __init__(self, root: str,
                 arguments_or_first_or_variable: Union[Sequence[Term],
                                                       Formula, str],
                 second_or_statement: Optional[Formula] = None):
        """Initializes a `Formula` from its root and root arguments, root
        operands, or root quantified variable name and statement.

        Parameters:
            root: the root for the formula tree.
            arguments_or_first_or_variable: the arguments for the root, if the
                root is a relation name or the equality relation; the first
                operand for the root, if the root is a unary or binary operator;
                the variable name to be quantified by the root, if the root is a
                quantification.
            second_or_statement: the second operand for the root, if the root is
                a binary operator; the statement to be quantified by the root,
                if the root is a quantification.
        """
        if is_equality(root) or is_relation(root):
            # Populate self.root and self.arguments
            assert isinstance(arguments_or_first_or_variable, Sequence) and \
                   not isinstance(arguments_or_first_or_variable, str)
            if is_equality(root):
                assert len(arguments_or_first_or_variable) == 2
            assert second_or_statement is None
            self.root, self.arguments = \
                root, tuple(arguments_or_first_or_variable)
        elif is_unary(root):
            # Populate self.first
            assert isinstance(arguments_or_first_or_variable, Formula)
            assert second_or_statement is None
            self.root, self.first = root, arguments_or_first_or_variable
        elif is_binary(root):
            # Populate self.first and self.second
            assert isinstance(arguments_or_first_or_variable, Formula)
            assert second_or_statement is not None
            self.root, self.first, self.second = \
                root, arguments_or_first_or_variable, second_or_statement
        else:
            assert is_quantifier(root)
            # Populate self.variable and self.statement
            assert isinstance(arguments_or_first_or_variable, str) and \
```

```
                is_variable(arguments_or_first_or_variable)
            assert second_or_statement is not None
            self.root, self.variable, self.statement = \
                root, arguments_or_first_or_variable, second_or_statement
```

The constructor of this class similarly creates an expression-tree representation for a (predicate-logic) formula. For example, given the term `my_term` defined in the example above, the data structure for representing the formula '(∃x[plus(s(x),3)=y]→GT(y,4))' is constructed using the following code:

```
my_formula = Formula('->',
                     Formula('E', 'x',
                             Formula('=', [my_term, Term('y')])),
                     Formula('GT', [Term('y'), Term('4')]))
```

Once again, note that the terms that serve as arguments of relation invocations (including arguments of the **equality relation**) are passed to the constructor together as a Python `list` rather than each argument separately (as is the case for, e.g., the operands of Boolean operators). Also note that for nullary relation invocations, this list will be of length zero. As with propositional formulas, to enable the safe reuse of existing formula and term objects as building blocks for (possibly even multiple) other formula and term objects, we have defined both of these classes to be **immutable** using the `@frozen` decorator.

As with propositional formulas, it is possible to represent predicate-logic terms and formulas as strings in a variety of notations, including infix, polish and reverse polish notations. We will use the representation defined above, which for terms is a functional notation that is similar to polish notation (only with added parentheses and commas since we have not defined the arity of each function name in advance), and for formulas is infix notation for Boolean operations and for equality, once again is a functional notation that is similar to polish notation (in the same sense, for the same reasons) for relation invocations, and is also similar to polish notation (with the addition of square brackets for readability) for quantifications.

Task 1. Implement the missing code for the method `__repr__()` of class `Term`, which returns a string that represents the term (in the usual functional notation defined above).

predicates/syntax.py

```
class Term:
        ⋮
    def __repr__(self) -> str:
        """Computes the string representation of the current term.

        Returns:
            The standard string representation of the current term.
        """
        # Task 7.1
```

Example: For the term `my_term` defined in the example above, `my_term.__repr__()` (and hence also `str(my_term)`) should return the string `'plus(s(x),3)'`.

Task 2. Implement the missing code for the method `__repr__()` of class `Formula`, which returns a string that represents the term (in the usual notation defined above—infix for Boolean operations and equality, functional for relation invocations, similar to polish for quantifications).

predicates/syntax.py

```
class Formula:
        ⋮
    def __repr__(self) -> str:
        """Computes the string representation of the current formula.

        Returns:
            The standard string representation of the current formula.
        """
        # Task 7.2
```

Example: For the formula `my_formula` defined in the example above, `my_formula.__repr__()` (and hence also `str(my_formula)`) should return the string `'(Ex[plus(s(x),3)=y]->GT(y,4))'`.

We observe that similarly to the string representations (both infix and prefix) of propositional formulas, the string representations of predicate-logic terms and formulas are also **prefix-free**, meaning that there are no two valid distinct predicate-logic terms such that the string representation of one is a prefix of the string representation of the other (with a small caveat, analogous to that of propositional formulas, that a variable name or constant name cannot be broken down so that only its prefix is taken), and similarly for predicate-logic formulas:

Lemma (Prefix-Free Property of Terms)**.** *No term is a prefix of another term, except for the case of a variable name as a prefix of another variable name, or a constant name as a prefix of another constant name.*

Lemma (Prefix-Free Property of Formulas)**.** *No formula is a prefix of another formula, except for the case of an equality with a variable name or constant name on the right-hand side, as a prefix of another equality with the same left-hand-side term as the former equality, but with another variable name or constant name (respectively) on the right-hand side (e.g.,* 'f(x)=a1' *as a prefix of* 'f(x)=a123'*).*

The proofs of the above lemmas are completely analogous to the proof of the lemma on the prefix-free property of propositional formulas from Chapter 1, since despite the richness of the language of Predicate Logic, we have taken care to define the various tokens in it—constant names, function names, variable names, relation names, quantifiers, etc.—so that they can be differentiated from one another based on their first character without the need to look beyond it (just like in Propositional Logic). Therefore, we omit these proofs (make sure that you can reconstruct their reasoning, though!). The prefix-free property allows for convenient parsing of predicate-logic expressions, using the same strategy that you followed for parsing propositional formulas in Chapter 1.

Task 3.

a. Implement the missing code for the static method `_parse_prefix(string)` of class `Term`, which takes a string that has a prefix that represents a term, and returns a term tree created from that prefix, and a string containing the unparsed remainder of the string (which may be empty, if the parsed prefix is in fact the entire string).

predicates/syntax.py

```
class Term:
        ⋮
    @staticmethod
    def _parse_prefix(string: str) -> Tuple[Term, str]:
        """Parses a prefix of the given string into a term.

        Parameters:
            string: string to parse, which has a prefix that is a valid
                representation of a term.

        Returns:
            A pair of the parsed term and the unparsed suffix of the string. If
            the given string has as a prefix a constant name (e.g., 'c12') or a
            variable name (e.g., 'x12'), then the parsed prefix will be that
            entire name (and not just a part of it, such as 'x1').
        """
        # Task 7.3a
```

Example: `Term._parse_prefix('s(x),3))')` should return a pair whose first element is a `Term` object equivalent to `Term('s', [Term('x')])`, and whose second element is the string `',3))'`.

Hint: Use recursion.

b. Implement the missing code for the static method `parse(string)` of class `Term`, which parses a given string representation of a term. You may assume that the input string represents a valid term.

predicates/syntax.py

```
class Term:
        ⋮
    @staticmethod
    def parse(string: str) -> Term:
        """Parses the given valid string representation into a term.

        Parameters:
            string: string to parse.

        Returns:
            A term whose standard string representation is the given string.
        """
        # Task 7.3b
```

Example: `Term.parse('plus(s(x),3))')` should return a `Term` object equivalent to `my_term` from the example above.

Hint: Use the `_parse_prefix()` method.

Similarly to the case of propositional formulas in Chapter 1, the reasoning and code that allowed you to implement the second (and the first) part of Task 3 without any ambiguity essentially prove the following theorem:

Theorem (Unique Readability of Terms)**.** *There is a unique derivation tree for every valid term in Predicate Logic.*

Task 4.

a. Implement the missing code for the static method `_parse_prefix(string)` of class `Formula`, which takes a string that has a prefix that represents a formula, and returns a formula tree created from that prefix, and a string containing the unparsed remainder of the string (which may be empty, if the parsed prefix is in fact the entire string).

predicates/syntax.py

```
class Formula:
        ⋮
    @staticmethod
    def _parse_prefix(string: str) -> Tuple[Formula, str]:
        """Parses a prefix of the given string into a formula.

        Parameters:
            string: string to parse, which has a prefix that is a valid
                representation of a formula.

        Returns:
            A pair of the parsed formula and the unparsed suffix of the string.
            If the given string has as a prefix a term followed by an equality
            followed by a constant name (e.g., 'f(y)=c12') or by a variable name
            (e.g., 'f(y)=x12'), then the parsed prefix will include that entire
            name (and not just a part of it, such as 'f(y)=x1').
        """
        # Task 7.4a
```

Example: `Formula._parse_prefix('Ex[plus(s(x),3)=y]->GT(y,4))')` should return a pair whose first element is a `Formula` equivalent to `Formula('E', 'x', Formula('=', [my_term, Term('y')]))` (for `my_term` from the example above), and whose second element is the string `'->GT(y,4))'`.

Hint: Use recursion, and use the `_parse_prefix()` method of class `Term` when needed.

b. Implement the missing code for the static method `parse(string)` of class `Formula`, which parses a given string representation of a formula. You may assume that the input string represents a valid formula.

predicates/syntax.py

```
class Formula:
        ⋮
    @staticmethod
    def parse(string: str) -> Formula:
        """Parses the given valid string representation into a formula.

        Parameters:
            string: string to parse.

        Returns:
            A formula whose standard string representation is the given string.
        """
        # Task 7.4b
```

Example: `Formula.parse('(Ex[plus(s(x),3)=y]->GT(y,4))')` should return a `Formula` object equivalent to `my_formula` from the example above.

Hint: Use the `_parse_prefix()` method.

Similarly to the case of terms, the reasoning and code that allowed you to implement the second (and the first) part of Task 4 without any ambiguity essentially prove the following theorem:

Theorem (Unique Readability of Formulas)**.** *There is a unique derivation tree for every valid formula in Predicate Logic.*

Completing the syntactic section of this chapter, you are now asked to implement a few methods that will turn out to be useful later on—methods that return all the constructs of a specific type (constant names, variable names, function names, or relation names) that are used in a term or a formula.

Task 5. Implement the missing code for the methods `constants()`, `variables()`, and `functions()` of class `Term`, which respectively return all of the constant names that appear in the term, all of the variable names that appear in the term, and all of the function names that appear in the term—each function name in a pair with the arity of its invocations. For the latter, you may assume that any function name that appears multiple times in the term is always invoked with the same arity.

predicates/syntax.py

```
class Term:
        ⋮
    def constants(self) -> Set[str]:
        """Finds all constant names in the current term.

        Returns:
            A set of all constant names used in the current term.
        """
        # Task 7.5a

    def variables(self) -> Set[str]:
        """Finds all variable names in the current term.

        Returns:
            A set of all variable names used in the current term.
        """
        # Task 7.5b

    def functions(self) -> Set[Tuple[str, int]]:
        """Finds all function names in the current term, along with their
        arities.

        Returns:
            A set of pairs of function name and arity (number of arguments) for
            all function names used in the current term.
        """
        # Task 7.5c
```

For formulas, it turns out that we will sometimes be interested in the set of **free** variable names in a formula rather than all variable names in it. A **free occurrence** of a

variable name in a formula is one that is not immediately next to a quantifier, nor **bound** by—that is, within the scope of—a quantification over this variable name. For example, in the formula '∀x[R(x,y)]', the occurrence of the variable name 'y' is free, but the occurrence of the variable name 'x' within 'R(x,y)' is not free since it is bound by the universal quantification '∀x'. To take a more delicate example, in the formula '(∃x[Q(x,y)]&x=0)', while the first occurrence of 'x' (not counting the one immediately next to the quantifier in '∃x') is bound (and therefore not free), the second one is free, and so the **free variable names**—the variable names that have free occurrences—in this formula are nonetheless 'x' and 'y'.

Task 6. Implement the missing code for the methods `constants()`, `variables()`, `free_variables()`, `functions()`, and `relations()` of class `Formula`, which respectively return all of the constants names that appear in the formula, all of the variable names that appear in the formula, all of the variable names that have free occurrences in the formula, all of the function names that appear in the formula (each function name in a pair with the arity of its invocations), and all of the relation names that appear in the formula—each relation name in a pair with the arity of its invocations. For the latter two, you may assume that any function name or relation name that appears multiple times in the formula is always invoked with the same arity.

predicates/syntax.py

```
class Formula:
        ⋮
    def constants(self) -> Set[str]:
        """Finds all constant names in the current formula.

        Returns:
            A set of all constant names used in the current formula.
        """
        # Task 7.6a

    def variables(self) -> Set[str]:
        """Finds all variable names in the current formula.

        Returns:
            A set of all variable names used in the current formula.
        """
        # Task 7.6b

    def free_variables(self) -> Set[str]:
        """Finds all variable names that are free in the current formula.

        Returns:
            A set of every variable name that is used in the current formula not
            only within a scope of a quantification on that variable name.
        """
        # Task 7.6c

    def functions(self) -> Set[Tuple[str, int]]:
        """Finds all function names in the current formula, along with their
        arities.

        Returns:
            A set of pairs of function name and arity (number of arguments) for
            all function names used in the current formula.
```

```
        """
        # Task 7.6d

    def relations(self) -> Set[Tuple[str, int]]:
        """Finds all relation names in the current formula, along with their
        arities.

        Returns:
            A set of pairs of relation name and arity (number of arguments) for
            all relation names used in the current formula.
        """
        # Task 7.6e
```

Finally, before moving on to the semantics of Predicate Logic, we note as we did for Propositional Logic that each **expression** (term or formula) is a finite-length string whose letters come from a finite number of characters, and thus there is a finite number of expressions of any given fixed length. We thus once again get the following simple fact:[7]

Theorem. *The set of terms and the set of formulas in Predicate Logic are both countably infinite.*

2 Semantics

We now move to the semantics of predicate-logic expressions (terms and formulas). Recall that in Propositional Logic both variables and formulas represent Boolean values, and a model directly specifies an **interpretation**, which is a Boolean value (a truth value), to each of the variable names, and the value of a formula is computed from the interpretations of its variable names according to the truth tables of the operators in the formula. As discussed above, in Predicate Logic our variables do not represent Boolean values, but rather values from some **universe** of elements—this is the grand set of values over which the quantifiers '∀' and '∃' quantify. In this logic, not only is the mapping from variable/constant names to values (from the universe) defined by the model, but in fact the universe that these values belong to is itself also defined by the model and can differ between models. Accordingly, the interpretations of all predicate-logic expression constructs that can be applied to terms—i.e., the mappings by which relation names and function names are interpreted—are also defined by the model.

The universe of a model could be any set (though we will restrict ourselves to finite sets in our code), for example the set of all elements of some field. Models in Predicate Logic define this universe and additionally provide interpretations of the constant names (as elements of the universe[8]), function names (as maps from tuples of elements of the universe to an element of the universe), and relation names (as maps from tuples of elements of the universe to Boolean values) used in the formulas. Thus, as already hinted to in the beginning of this chapter, the semantic interpretation (value) that a model induces on a term will be an element from the universe, while the value induced by a model on a formula

[7] As in Propositional Logic, all of our results will extend naturally via analogous proofs to allow for sets of constant/variable/function/relation names of arbitrary cardinality, which would imply also terms and formulas sets of arbitrary cardinality. As before, in the few places where the generalization will not be straightforward, we will explicitly discuss this.

[8] In this sense, somewhat confusingly, constants in Predicate Logic are analogous to variables in Propositional Logic, while as we will see below variables in Predicate Logic play a different role that has to do with quantifications and has no analogue in Propositional Logic.

will be Boolean (a truth value). Variables are not assigned a specific interpretation by a model, and as we will see below, will only get a value once it is assigned to them by an additional assignment—corresponding to a quantification—that complements the model (as a variable name is syntactically a term, the value assigned to a variable name will also be an element of the universe of the model).

Definition (Model)**.** A **model** in predicate logic consists of a set of elements Ω called the **universe** of the model, as well as an **interpretation** for a set of constant names, function names, and relation names. An **interpretation** of a constant name is an element in the universe Ω, an **interpretation** of an n-ary function name is a function $f : \Omega^n \to \Omega$, and an **interpretation** of an n-ary relation name is a subset of Ω^n (for which the relation holds[9]).

For example, the following describes the five-element field F_5 (sometimes also denoted as $\mathbb{Z}_5$ or $GF(5)$) as a model for a field:

- Universe $\Omega = \{0, 1, 2, 3, 4\}$.
- The interpretation of the constant name '0' is $0 \in \Omega$. The interpretation of the constant name '1' is $1 \in \Omega$.
- The interpretation of the binary function name 'plus' is addition modulo 5: $\text{plus}(0, 0) = 0$, $\text{plus}(0, 1) = 1$, ..., $\text{plus}(0, 4) = 4$, $\text{plus}(1, 0) = 1$, ..., $\text{plus}(1, 3) = 4$, $\text{plus}(1, 4) = 0$, ..., $\text{plus}(4, 4) = 3$. The interpretation of the binary function name 'times' is multiplication modulo 5, for example, $\text{times}(3, 3) = 4$.
- The interpretation of the unary relation name[10] 'IsPrimitive' is $\{(2), (3)\}$.
- Depending on the expressions that we wish to evaluate, we could also add corresponding interpretations for additional function names and relation names, such as the unary function names 'inverse' and 'multiplicativeInverse'.

The file `predicates/semantics.py` defines a class `Model` that holds a semantic model for predicate-logic expressions.

predicates/semantics.py

```
#: A generic type for a universe element in a model.
T = TypeVar('T')

@frozen
class Model(Generic[T]):
    """An immutable model for predicate-logic constructs.

    Attributes:
        universe: the set of elements to which terms can be evaluated and over
            which quantifications are defined.
        constant_interpretations: mapping from each constant name to the
            universe element to which it evaluates.
```

[9]While this standard representation of a relation as the set of all n-tuples of universe elements for which the relation holds (i.e., is true) will be slightly more convenient for us in our code, and is also more aesthetic, we will at times continue to think of relations also as functions from n-tuples of universe elements to Boolean values when the analogy to functions (from n-tuples of universe elements to universe elements) is intuitively useful.

[10]A field element is called primitive if it *generates* the *multiplicative group* of the field, that is, if every nonzero element of the field can be written as an integer power of that element.

```
        relation_arities: mapping from each relation name to its arity, or to
            ``-1`` if the relation is the empty relation.
        relation_interpretations: mapping from each n-ary relation name to
            argument n-tuples (of universe elements) for which the relation is
            true.
        function_arities: mapping from each function name to its arity.
        function_interpretations: mapping from each n-ary function name to the
            mapping from each argument n-tuple (of universe elements) to the
            universe element that the function outputs given these arguments.
    """
    universe: FrozenSet[T]
    constant_interpretations: Mapping[str, T]
    relation_arities: Mapping[str, int]
    relation_interpretations: Mapping[str, AbstractSet[Tuple[T, ...]]]
    function_arities: Mapping[str, int]
    function_interpretations: Mapping[str, Mapping[Tuple[T, ...], T]]

    def __init__(self, universe: AbstractSet[T],
                 constant_interpretations: Mapping[str, T],
                 relation_interpretations:
                 Mapping[str, AbstractSet[Tuple[T, ...]]],
                 function_interpretations:
                 Mapping[str, Mapping[Tuple[T, ...], T]] = frozendict()):
        """Initializes a `Model` from its universe and constant, relation, and
        function name interpretations.

        Parameters:
            universe: the set of elements to which terms are to be evaluated
                and over which quantifications are to be defined.
            constant_interpretations: mapping from each constant name to a
                universe element to which it is to be evaluated.
            relation_interpretations: mapping from each relation name that is to
                be the name of an n-ary relation, to the argument n-tuples (of
                universe elements) for which the relation is to be true.
            function_interpretations: mapping from each function name that is to
                be the name of an n-ary function, to a mapping from each
                argument n-tuple (of universe elements) to a universe element
                that the function is to output given these arguments.
        """
        for constant in constant_interpretations:
            assert is_constant(constant)
            assert constant_interpretations[constant] in universe
        relation_arities = {}
        for relation in relation_interpretations:
            assert is_relation(relation)
            relation_interpretation = relation_interpretations[relation]
            if len(relation_interpretation) == 0:
                arity = -1 # any
            else:
                some_arguments = next(iter(relation_interpretation))
                arity = len(some_arguments)
                for arguments in relation_interpretation:
                    assert len(arguments) == arity
                    for argument in arguments:
                        assert argument in universe
            relation_arities[relation] = arity
        function_arities = {}
        for function in function_interpretations:
```

```
            assert is_function(function)
            function_interpretation = function_interpretations[function]
            assert len(function_interpretation) > 0
            some_argument = next(iter(function_interpretation))
            arity = len(some_argument)
            assert arity > 0
            assert len(function_interpretation) == len(universe)**arity
            for arguments in function_interpretation:
                assert len(arguments) == arity
                for argument in arguments:
                    assert argument in universe
                assert function_interpretation[arguments] in universe
            function_arities[function] = arity

        self.universe = frozenset(universe)
        self.constant_interpretations = frozendict(constant_interpretations)
        self.relation_arities = frozendict(relation_arities)
        self.relation_interpretations = \
            frozendict({relation: frozenset(relation_interpretations[relation])
                        for relation in relation_interpretations})
        self.function_arities = frozendict(function_arities)
        self.function_interpretations = \
            frozendict({function: frozendict(function_interpretations[function])
                        for function in function_interpretations})
```

You have probably guessed by now how we would evaluate, in a model, terms that are simply constant names, and how we would recursively evaluate terms that are function invocations. What about variable names, though? A model specifies no interpretation for variable names. We evaluate variable names in the following way, which may seem a bit cryptic right now, but will become clearer once we talk about evaluating quantified formulas: a term that contains variable names has a defined value in a given model M for this term only once we additionally define an **assignment** A that assigns an element in the universe of M to each variable name in the term.

Definition (Value of Term in Model)**.** Given a term τ, a model M with interpretations for (at least) the constant and function names of τ, and an assignment A for (at least) the variable names of τ, the **value** of the term τ in the model M under the assignment A is an element in the universe Ω of M that we define recursively:

- If τ is a constant name c, then its value is given directly by the model M as the interpretation of this constant name.
- If τ is a variable name x, then its value is given directly by the assignment A.
- If τ is an n-ary function invocation $f(t_1,\ldots,t_n)$, then its value is the result of applying the interpretation of the function name f (which is a function from Ω^n to Ω that is given directly by the model) to the (recursively defined) values of its arguments $t_1,\ldots,t_n$ in M under A.

For example, the value of the term 'times(x,plus(1,1))' in the above field model under the assignment that assigns the value $4 \in \Omega$ to the variable name 'x', is $3 \in \Omega$. If a term has no variable names, then an assignment is not required in order to fully evaluate it in a model using the above definition (that is, it has the same value in this model under any assignment). For example, the value of the term 'plus(plus(plus(plus(1,1),1),1),1)' in the above field model (under any assignment) is $0 \in \Omega$.

Task 7. Implement the missing code for the method `evaluate_term(term, assignment)` of class `Model`, which returns the value (in the universe of the model) of the given term under the given assignment of values to its variable names.

predicates/semantics.py

```
class Model:
        ⋮
    def evaluate_term(self, term: Term,
                      assignment: Mapping[str, T] = frozendict()) -> T:
        """Calculates the value of the given term in the current model under the
        given assignment of values to variable names.

        Parameters:
            term: term to calculate the value of, for the constant and function
                names of which the current model has interpretations.
            assignment: mapping from each variable name in the given term to a
                universe element to which it is to be evaluated.

        Returns:
            The value (in the universe of the current model) of the given
            term in the current model under the given assignment of values to
            variable names.
        """
        assert term.constants().issubset(self.constant_interpretations.keys())
        assert term.variables().issubset(assignment.keys())
        for function,arity in term.functions():
            assert function in self.function_interpretations and \
                   self.function_arities[function] == arity
        # Task 7.7
```

Similarly, given a model, a predicate-logic formula, and an assignment for the free variable names of the formula, we can recursively associate a Boolean truth value with the formula:

Definition (Truth Value of Formula in Model)**.** Given a formula ϕ, a model M with interpretations for (at least) the constant, function, and relation names of ϕ, and an assignment for (at least) the free variable names of ϕ, we define the **(truth) value** of the formula ϕ in the model M under the assignment A recursively:

- If ϕ is an equality 't_1=t_2', then its value is *True* if and only if in M under A the value of the term t_1 is the same element of the universe Ω of M as the value of the term t_2 (where these values, each an element of Ω, are as recursively defined above).
- If ϕ is an n-ary relation invocation '$R(t_1,\ldots,t_n)$', then its value is *True* if and only if the tuple of the values of these terms (ordered from 1 to n) in M under A is contained in the interpretation of the relation name R in M.
- Unary and binary Boolean operations are evaluated as in propositional formulas (see Chapter 2), with the subformulas evaluated also in M under A.
- If ϕ is a universally quantified formula '$\forall x[\phi]$', then its value is *True* if and only if ϕ evaluates to *True* in M under *every* assignment created from A by assigning some element in Ω to the variable name x (this determines the values of the free occurrences of x in ϕ) while leaving the values assigned to other variable names as in

A; if ϕ is an existentially quantified formula '$\exists x[\phi]$', then its value is *True* if and only if ϕ evaluates to *True* in M under *some* (that is, one or more) assignment created from A by assigning some element in Ω to the variable name x (this determines the values of the free occurrences of x in ϕ) while leaving the values assigned to other variable names as in A.

Note that analogously to the case of terms, a formula only has a defined value in a given model once we additionally define an assignment that gives a value to each of its *free* variable names (such an assignment is needed when evaluating any term in this formula that contains occurrences of variable names that are free occurrences with respect to the full formula). For example, since in the formula '∀x[times(x,y)=x]' the variable name 'y' is free, this formula only has a defined truth value if an additional assignment assigns a value to 'y', while the variable name 'x' need not be explicitly assigned a value by the assignment because by definition, since it is bound by a universal quantification, we go over all possible values for it when evaluating the formula. The truth value of this formula in the above field model is *True* under an assignment that assigns the value $1 \in \Omega$ to the variable name 'y', and *False* under any assignment that assigns any other value (in $\{0, 2, 3, 4\}$) to 'y'.

Analogously to terms, if a formula has no free variable names, then an assignment is not required in order to fully evaluate it in a model using the above definition (that is, it has the same value in the model under any assignment). For example, the truth value of the formula '∀x[∃y[times(x,y)=1]]' in the above field model (under any assignment) is *False*, while that of the formula '∀x[(x=0|∃y[times(x,y)=1])]' is *True*, and so is that of the formula '∀x[plus(plus(plus(plus(x,x),x),x),x)=0]'. Similarly, in that model the truth value of the formula '∀x[(x=0|IsPrimitive(x))]' is *False*, and that of the formula '∀x[(IsPrimitive(x)→∀y[(y=0|(y=x|(y=times(x,x)|(y=times(times(x,x),x)|y=times(times(times(x,x),x),x)))))])]' is *True*.

Note that the above definitions imply that we interpret an occurrence of a variable name in any specific context by the innermost **scope** containing it that defines an assignment for it: the innermost quantification if such exists, or the given assignment (which in this sense defines an assignment for the "global scope" of the entire formula that is evaluated) if there exists no such quantification (i.e., if that occurrence of the variable name is free). Thus, in the (algebraically nonsensical) formula '(x=0|(∃x[IsPrimitive(x)→∀x[times(x,x)=1]]))', both occurrences of 'x' in 'times(x,x)' are **bound by** (i.e., when evaluated, will get their value from the assignment corresponding to) the universal quantifier, while the occurrence of 'x' in 'IsPrimitive(x)' is bound by the existential quantifier and the occurrence of 'x' in 'x=0' is free and its value must be given by an assignment in order to evaluate this formula. Note that the subformula '(∃x[IsPrimitive(x)→∀x[times(x,x)=1]])' can be evaluated in a model without requiring any value to be given to 'x' by an assignment, since there are no free occurrences of 'x' in it.

Task 8. Implement the missing code for the method `evaluate_formula(formula, assignment)` of class `Model`, which returns the truth value of the given formula under the given assignment of values to its free variable names.

predicates/semantics.py

```
class Model:
        ⋮
    def evaluate_formula(self, formula: Formula,
                         assignment: Mapping[str, T] = frozendict()) -> bool:
        """Calculates the truth value of the given formula in the current model
        under the given assignment of values to free occurrences of variable
```

```
        names.

        Parameters:
            formula: formula to calculate the truth value of, for the constant,
                function, and relation names of which the current model has
                interpretations.
            assignment: mapping from each variable name that has a free
                occurrence in the given formula to a universe element to which
                it is to be evaluated.

        Returns:
            The truth value of the given formula in the current model under the
            given assignment of values to free occurrences of variable names.
        """
        assert formula.constants().issubset(
            self.constant_interpretations.keys())
        assert formula.free_variables().issubset(assignment.keys())
        for function,arity in formula.functions():
            assert function in self.function_interpretations and \
                   self.function_arities[function] == arity
        for relation,arity in formula.relations():
            assert relation in self.relation_interpretations and \
                   self.relation_arities[relation] in {-1, arity}
        # Task 7.8
```

Finally, as we will see we will many times be interested in checking whether a formula evaluates to *True* in a given model under *all* possible assignments to its free variable names. In this case, as in Propositional Logic, we will say that this model **is a model of** this formula. We will furthermore be interested in evaluating like this not merely a single formula but a whole set of formulas. The last task of this chapter is to implement a helper method that performs this.

Task 9. Implement the missing code for the method `is_model_of(formulas)` of class `Model`, which returns whether the model is a model of each of the given formulas, regardless of which values from the universe of the model are assigned to its free variable names.

predicates/semantics.py

```
class Model:
        ⋮
    def is_model_of(self, formulas: AbstractSet[Formula]) -> bool:
        """Checks if the current model is a model of the given formulas.

        Parameters:
            formulas: formulas to check, for the constant, function, and
                relation names of which the current model has interpretations.

        Returns:
            ``True`` if each of the given formulas evaluates to true in the
            current model under any assignment of elements from the universe of
            the current model to the free occurrences of variable names in that
            formula, ``False`` otherwise.
        """
        for formula in formulas:
            assert formula.constants().issubset(
                self.constant_interpretations.keys())
            for function,arity in formula.functions():
                assert function in self.function_interpretations and \
```

```
                    self.function_arities[function] == arity
            for relation,arity in formula.relations():
                assert relation in self.relation_interpretations and \
                    self.relation_arities[relation] in {-1, arity}
        # Task 7.9
```