

Chapter 7:

First-Order Predicate Logic: Syntax and Semantics

The propositional logic that we studied up to this point, in the first half of this book, is not rich enough by itself to represent many common logical statements. Consider for example the simple **syllogism**: All Greeks are Men and there exist Greeks; therefore, there exist Men. Propositional logic could not 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, **first-order logic** or **predicate logic**). This logic will be strong enough to express and formalize such notions.

For example the above statement can be written in predicate logic as

$$‘((\forall x[(\text{Greek}(x) \rightarrow \text{Man}(x))] \& \exists x[\text{Greek}(x)]) \rightarrow \exists x[\text{Man}(x)])’$$

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 $\forall x[\forall y[x+y=y+x]]$, which we will actually write in the slightly more cumbersome notation $\forall x[\forall y[\text{plus}(x,y)=\text{plus}(y,x)]]$. 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, 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 propositional logic that we were analyzing 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 first-order predicate logic.

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 “formulas,” called **terms**, that represent a not-necessarily-Boolean object, such as ‘plus(x,y)’ or even ‘x’. As with propositional logic, we start by syntactically defining terms and formulas in first-order predicate logic, and will only later give them semantic meaning. Nonetheless, remembering that formulas will eventually represent Boolean statements while terms will represent not-necessarily-Boolean objects 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 first-order predicate logic.

Definition (Term). The following strings are valid **terms** in first-order logic:

- A **variable name**: a sequence of alphanumeric characters that begins with a letter in ‘u’...‘z’.
Examples: ‘x’, ‘y12’, ‘zLast’.
- A **constant name**: a sequence of alphanumeric characters that begins with a digit or with a letter in ‘a’...‘d’; or an underscore (with nothing before or after it).
Examples: ‘0’, ‘c1’, ‘7x’, ‘_’.
- An n -ary **function invocation** of the form ‘ $f(t_1, \dots, t_n)$ ’, where f is a **function name** denoted by a sequence of alphanumeric characters that begins with a letter in ‘f’...‘t’, where¹ $n \geq 1$, and where each t_i is itself a valid term.
Examples: ‘plus(x,y)’, ‘s(s(0))’, ‘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(s: str) -> bool:
    """Checks if the given string is a constant name.

    Parameters:
        s: string to check.

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

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

    Parameters:
        s: string to check.

    Returns:
        ``True`` if the given string is a variable name, ``False`` otherwise.

```

¹Another 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.

```

    """
    return s[0] >= 'u' and s[0] <= 'z' and s.isalnum()

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

    Parameters:
        s: string to check.

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

@frozen
class Term:
    """An immutable first-order 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 to 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) -> None:
        """Initializes a `Term` from its root and root arguments.

        Parameters:
            root: the root for the formula tree.
            arguments: the arguments to 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
            self.root = root
            self.arguments = tuple(arguments)
            assert len(self.arguments) > 0

```

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:

```
t = 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 first-order predicate logic**.

Definition (Formula). The following strings are valid **formulas** in first-order logic:

- An **equality** of the form ' $t_1=t_2$ ', where each of t_1 and t_2 is a valid terms.
Examples: ' $0=0$ ', ' $s(0)=1$ ', ' $\text{plus}(x,y)=\text{plus}(y,x)$ '.
- An n -ary **relation invocation** of the form ' $R(t_1, \dots, t_n)$ ', 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 valid term.
Examples: ' $R(x,y)$ ', ' $\text{Plus}(s(0),x,s(x))$ ', ' $Q()$ '.
- A **unary negation** of the form ' $\sim\phi$ ', where ϕ is a valid formula.
- A **binary operation** of the form ' $(\phi*\psi)$ ', where $*$ is one of the binary operators '|', '&', or ' \rightarrow ',² and each of ϕ and ψ is a valid 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 valid formula.
Examples: ' $\forall x[x=x]$ ', ' $\exists x[R(7,y)]$ ', ' $\forall x[\exists y[R(x,y)]]$ ', ' $\forall x[(R(x)|\exists x[Q(x)])]$ '.

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

    Parameters:
        s: string to check.

    Returns:
        ``True`` if the given string is the equality relation, ``False``
        otherwise.
    """
    return s == '='

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

    Parameters:
        s: string to check.

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

```

²We could have again used any other (larger or smaller) set of complete Boolean operators 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 these three operators as we did in the first half of this book.

```

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

    Parameters:
        s: string to check.

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

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

    Parameters:
        s: string to check.

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

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

    Parameters:
        s: string to check.

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

@frozen
class Formula:
    """An immutable first-order formula in tree representation, composed from
    relation names applied to first-order 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 to the root, if the root is a relation name or
            the equality relation.
        first: the first operand to the root, if the root is a unary or binary
            operator.
        second: the second operand to the root, if the root is a binary
            operator.
        variable: the variable name quantified by the root, if the root is a
            quantification.
        predicate: the predicate 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]
predicate: Optional[Formula]

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

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

```

The constructor of this class similarly creates an expression-tree representation for a formula. For example, given the term t defined in the example above, the data structure for representing the formula $(\exists x[\text{plus}(s(x), 3) = y] \rightarrow \text{GT}(y, 4))$ is constructed using the following code:

```

f = Formula('->', Formula('E', 'x', Formula('=', [t, 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 relations, this list will be of length zero. As with proposition 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 first-order formulas and terms 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 prefix 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, and once again is a functional notation that is similar to prefix notation (in the same sense, for the same reasons) for relation invocations.

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 `t` defined in the example above, `t.__repr__()` (and hence also `str(t)`) 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).

```

----- 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 `f` defined in the example above, `f.__repr__()` (and hence also `str(f)`) 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 first-order terms and first-order formulas

are also **prefix-free**, meaning that there are no two valid distinct first-order 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 first-order 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 term on the left-hand side and with another variable name or constant name (respectively) on the right-hand side.*

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, so we omit these proofs (make sure that you can reconstruct their reasoning, thought!). The prefix-free property allows for convenient parsing of first-order expressions, using the same strategy that you followed for parsing propositional formulas in Chapter 1.

Task 3 (Parsing Terms).

1. Implement the missing code for the static method `parse_prefix(s)` 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(s: str) -> Tuple[Term, str]:
        """Parses a prefix of the given string into a term.

        Parameters:
            s: 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 7.3.1

```

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.

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

```

----- predicates/syntax.py -----
class Term:
    :
    @staticmethod
    def parse(s: str) -> Term:
        """Parses the given valid string representation into a term.

        Parameters:
            s: string to parse.

        Returns:
            A term whose standard string representation is the given string.
        """
        # Task 7.3.2

```

Example: `Term.parse('plus(s(x),3)')` should return a `Term` object equivalent to `t` 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 proves the following theorem.

Theorem (Unique Readability of Terms). *There is a unique derivation tree for every valid term in first-order predicate logic.*

Task 4 (Parsing Formulas).

1. Implement the missing code for the static method `parse_prefix(s)` 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(s: str) -> Tuple[Formula, str]:
        """Parses a prefix of the given string into a formula.

        Parameters:
            s: 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., 'c12') or by a variable name
            (e.g., 'x12'), then the parsed prefix will include that entire name
            (and not just a part of it, such as 'x1').

```

```
"""
# Task 7.4.1
```

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('=', [t, Term('y')]))` (for `t` 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.

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

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

        Parameters:
            s: string to parse.

        Returns:
            A formula whose standard string representation is the given string.
        """
    # Task 7.4.2
```


Example: `Formula.parse('(Ex[plus(s(x),3)=y]->GT(y,4))')` should return a `Formula` object equivalent to `f` 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 proves the following theorem.

Theorem (Unique Readability of Formulas). *There is a unique derivation tree for every valid formula in first-order 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 elements of a specific type (constants, variables, functions, or relations) 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 constants that appear in the term, all of the variables that appear in the term, and all of the functions that appear in the term—each function in a pair with the arity  its invocations. For the latter, you can 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 7.5.1

    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 7.5.2

    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 7.5.3

```

For formulas, it turns out that we will sometimes be interested in the set of **free** variables in a formula rather than all variables in it. A **free occurrence** of a variable 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. For example, in the formula $\forall x[R(x,y)]$, the variable y is free, but the occurrence of the variable x within $R(x,y)$ is not free since it is bound by the universal quantification $\forall x$. To take a more delicate example, in the formula $(\exists x[Q(x,y)] \& x=0)$, while the first occurrence of x (not counting the x immediately next to the quantifier in $\exists x$) is bound (and therefore not free), the second one is free, and so the free variables 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 that appear in the formula, all of the variables that appear in the formula, all of the variables that have free occurrences in the formula, all of the functions that appear in the formula (each function in a pair with the arity of its invocations), and all of the relations that appear in the formula—each relation in a pair with the arity of its invocations. For the latter two, you can 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 7.6.1

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 7.6.2

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

    Returns:
        A set of all variable names used in the current formula not only
        within a scope of a quantification on those variable names.
    """
    # Task 7.6.3

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 7.6.4

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 7.6.5

```

Finally, before moving on to the semantics of first-order 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 formulas of any given fixed length. We thus once again get the following simple fact:³

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

³As in propositional logic, all of our results will extend naturally via analogous proofs to allow for sets of constants/variables/functions/relations 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.

2 Semantics

We now move to the semantics of first-order expressions (terms and formulas). Recall that in propositional logic both variables and formulas represent Boolean values, where a model directly gives a **meaning** that is a Boolean value to the variables, and the value of a formula is computed from the meanings of its variables according to the truth tables of the operators in the formula. As discussed above, in first-order logic predicate variables do not represent Boolean values, but rather get a value from some **universe** of elements—the grand set of values over which the quantifiers ‘ \forall ’ and ‘ \exists ’ quantify. In this logic, not only is the mapping from variables/constants 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 meanings of all first-order expression constructs that can be applied to variables—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 first-order logic define this universe and additionally provide meanings to the constants (as elements of the universe⁴), functions (as maps from tuples of elements of the universe to an element of the universe), and relations (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 meaning (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 will be Boolean. Free variables are not assigned a specific meaning 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 is syntactically a term, the value assigned to a variable will also be an element of the universe of the model).

Definition (Model). A **model** for a first-order expression consists of a set of elements Ω called the **universe**, as well as a **meaning** for each constant name, function name, and relation name in the expression. A **meaning** of a constant name is an element in the universe Ω , a **meaning** of an n -ary relation name is a subset of Ω^n (for which the relation holds), and a **meaning** of an n -ary function name is a function $f : \Omega^n \rightarrow \Omega$.

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

- $\Omega = \{0, 1, 2, 3, 4\}$.
- The meaning of the constant ‘0’ is $0 \in \Omega$. The meaning of the constant ‘1’ is $1 \in \Omega$.
- The meaning of the binary function ‘plus’ is addition modulo 5: $\text{plus}(0, 0) = 0$, $\text{plus}(0, 1) = 1$, \dots , $\text{plus}(0, 4) = 4$, $\text{plus}(1, 0) = 1$, \dots , $\text{plus}(1, 3) = 4$, $\text{plus}(1, 4) = 0$, \dots , $\text{plus}(4, 4) = 3$. The meaning of the binary function ‘times’ is multiplication modulo 5, for example, $\text{times}(3, 3) = 4$.
- The meaning of the unary relation⁵ ‘IsPrimitive’ is $\{2, 3\}$.

⁴In this sense, somewhat confusingly, constants in first-order logic are analogous to variables in propositional logic, while as we will see below variables in first-order logic play a different role that has to do with quantifications and has no analogue in propositional logic.

⁵A 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.

- Depending on the elements that we wish to evaluate, we could also add corresponding meanings for additional functions and relations, such as the unary functions ‘inverse’ and ‘multiplicativeInverse’.

The file `predicates/semantics.py` defines a class `Model` that holds a semantic model for first-order 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 first-order logic constructs.

    Attributes:
        universe: the set of elements to which terms can be evaluated and over
            which quantifications are defined.
        constant_meanings: mapping from each constant name to the universe
            element to which it evaluates.
        relation_aritys: mapping from each relation name to the arity of the
            relation, or to ``-1`` if the relation is the empty relation.
        relation_meanings: mapping from each n-ary relation name to argument
            n-tuples (of universe elements) for which the relation is true.
        function_aritys: mapping from each function name to the arity of the
            function.
        function_meanings: 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_meanings: Mapping[str, T]
    relation_aritys: Mapping[str, int]
    relation_meanings: Mapping[str, AbstractSet[Tuple[T, ...]]]
    function_aritys: Mapping[str, int]
    function_meanings: Mapping[str, Mapping[Tuple[T, ...], T]]

    def __init__(self, universe: AbstractSet[T],
                  constant_meanings: Mapping[str, T],
                  relation_meanings: Mapping[str, AbstractSet[Tuple[T, ...]]],
                  function_meanings: Mapping[str, Mapping[Tuple[T, ...], T]] =
                      frozendict()) -> None:
        """Initializes a `Model` from its universe and constant, relation, and
        function meanings.

    Parameters:
        universe: the set of elements to which terms are to be evaluated
            and over which quantifications are to be defined.
        constant_meanings: mapping from each constant name to a universe
            element to which it is to be evaluated.
        relation_meanings: 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_meanings: 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.

    """

```

```

self.universe = frozenset(universe)

for constant in constant_meanings:
    assert is_constant(constant)
    assert constant_meanings[constant] in universe
self.constant_meanings = frozendict(constant_meanings)

relation_arities = {}
for relation in relation_meanings:
    assert is_relation(relation)
    relation_meaning = relation_meanings[relation]
    if len(relation_meaning) == 0:
        arity = -1 # any
    else:
        some_arguments = next(iter(relation_meaning))
        arity = len(some_arguments)
        for arguments in relation_meaning:
            assert len(arguments) == arity
            for argument in arguments:
                assert argument in universe
    relation_arities[relation] = arity
self.relation_meanings = \
    frozendict({relation: frozenset(relation_meanings[relation]) for
                relation in relation_meanings})
self.relation_arities = frozendict(relation_arities)

function_arities = {}
for function in function_meanings:
    assert is_function(function)
    function_meaning = function_meanings[function]
    assert len(function_meaning) > 0
    some_argument = next(iter(function_meaning))
    arity = len(some_argument)
    assert arity > 0
    assert len(function_meaning) == len(universe)**arity
    for arguments in function_meaning:
        assert len(arguments) == arity
        for argument in arguments:
            assert argument in universe
        assert function_meaning[arguments] in universe
    function_arities[function] = arity
self.function_meanings = \
    frozendict({function: frozendict(function_meanings[function]) for
                function in function_meanings})
self.function_arities = frozendict(function_arities)

```

Definition (Truth Value of Term in Model). A term that contains no variables has an **induced value** in any given model M for this term. This induced value is an element in Ω :

- The induced value of a constant name c is given directly by the model as the meaning of this constant name.
- The induced value of an n -ary function invocation $f(t_1, \dots, t_n)$ is recursively defined by applying the meaning of the function f (which is a function from Ω^n to Ω that is given directly by the model) to the (recursively defined) induced values of its arguments t_1, \dots, t_n .

For example, the induced value of the term ‘plus(plus(plus(plus(1,1),1),1),1)’ in the above field model is $0 \in \Omega$.

A term that does contain variable names has an induced value in a given model M for this term only once we additionally define an **assignment** A that consists of an element in Ω for each variable name in the term.

- The induced value of a variable name x in a given assignment is given directly by the assignment.

For example, the induced value of the term ‘times(x,plus(1,1))’ in the above field model, with the assignment that assigns the value $4 \in \Omega$ to the variable ‘x’, is $3 \in \Omega$.

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

```

----- 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, for the
        given assignment of values to variables names.

        Parameters:
            term: term to calculate the value of, for the constants and
                  functions of which the current model has meanings.
            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, for the given assignment of values to
            variable names.
        """
        assert term.constants().issubset(self.constant_meanings.keys())
        assert term.variables().issubset(assignment.keys())
        for function, arity in term.functions():
            assert function in self.function_meanings and \
                   self.function_arities[function] == arity
        # Task 7.7

```

Similarly, given a model for a first-order formula and an assignment to variables names, we can recursively associate an induced Boolean truth value with the formula:

Definition (Induced Value of Formula in Model). Let M be a model and let A be an assignment.

- The induced truth value of an equality ‘ $t_1=t_2$ ’ is *True* if and only if the induced value of term t_1 is the same element of Ω as the induced value of t_2 (where these values, each an element of Ω , are as recursively defined above).
- The induced truth value of an n -ary relation invocation ‘ $R(t_1, \dots, t_n)$ ’ is *True* if and only if the tuple of the induced values of these terms (ordered from 1 to n) is contained in the meaning of the relation name R .

- Unary and binary Boolean operations are evaluated as in propositional formulas. (See Chapter 2.)
- The induced truth value of a universally quantified formula ' $\forall x[\phi]$ ' is *True* if and only if ϕ evaluates to *True* for *every* assignment of a value from Ω that can be added to A for x (this assignment determines the values of the free occurrences of x in ϕ), while the induced truth value of an existentially quantified formula ' $\exists x[\phi]$ ' is *True* if and only if ϕ evaluates to *True* for *some* (that is, one or more) assignment of a value from Ω that can be added to A for x (this assignment determines the values of the free occurrences of x in ϕ).

For example, the induced truth value of the formula ' $\forall x[\exists y[\text{times}(x,y)=1]]$ ' in the above field model is *False*, while that of the formula ' $\forall x[(x=0|\exists y[\text{times}(x,y)=1])]$ ' is *True*, and so is that of the formula ' $\forall x[\text{times}(\text{times}(\text{times}(x,x),x),x)=1]$ '. Similarly, in that model the induced truth value of the formula ' $\forall x[(x=0|\text{IsPrimitive}(x))]$ ' is *False*, and that of the formula ' $\forall x[(\text{IsPrimitive}(x) \rightarrow \forall y[(y=0|(y=x|(y=\text{times}(x,x)|(y=\text{times}(\text{times}(x,x),x)|y=\text{times}(\text{times}(\text{times}(x,x),x),x))))))]]]$ ' is *True*.

Note that a formula only has an induced value in a given model once we additionally define an assignment that gives a value to each of its *free* variables (such an assignment is needed when evaluating any term in this formula that contains occurrences of variables that are free occurrences with respect to the full formula). For example, since in the formula ' $\forall x[\text{times}(x,y)=x]$ ' the variable y is free, this formula only has a defined induced truth value if an additional assignment assigns a value to y , while the variable 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 induced truth value of this formula in the above field model is *True* for the assignment that assigns the value $1 \in \Omega$ to the variable ' y ' and *False* for any assignment that assigns any other value (in $\{0, 2, 3, 4\}$) to ' y '.

Note that the above definitions imply that we interpret an occurrence of a variable name in any specific context by the innermost **scope** surrounding it that defines an assignment for it: the innermost quantification if such exists, or the given assignment if there exists no such quantification (i.e., if that occurrence of the variable name is free). Thus, in the (algebraically nonsensical) formula ' $(x=0|(\exists x[\text{IsPrimitive}(x) \rightarrow \forall x[\text{times}(x,x)=1]]))$ ', both occurrences of ' x ' in ' $\text{times}(x,x)$ ' are quantified by the universal quantifier, while the ' x ' in ' $\text{IsPrimitive}(x)$ ' is quantified by the existential quantifier and the ' 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 ' $(\exists x[\text{IsPrimitive}(x) \rightarrow \forall x[\text{times}(x,x)=1]])$ ' has a value in any model for it 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 induced truth value of the given formula, with the given assignment of values to its free variables.

```

_____ 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,

```

```

for the given assignment of values to free occurrences of variables
names.

Parameters:
    formula: formula to calculate the truth value of, for the constants,
            functions, and relations of which the current model has
            meanings.
    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, for the
    given assignment of values to free occurrences of variable names.
"""
assert formula.constants().issubset(self.constant_meanings.keys())
assert formula.free_variables().issubset(assignment.keys())
for function,arity in formula.functions():
    assert function in self.function_meanings and \
           self.function_arities[function] == arity
for relation,arity in formula.relations():
    assert relation in self.relation_meanings and \
           self.relation_arities[relation] in {-1, arity}
# Task 7.8

```

Finally, as we will see we will many times be interested in evaluating in a given model not merely a single formula but a whole set of formulas, and checking whether *each of them* evaluates to *True* in the model for *all* possible assignments to its free variables. 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 valid model of each of the given formulas, regardless of which values from the universe of the model are assigned to its free variables.

```

----- predicates/semantics.py -----
class Model:
    :
    :
    def is_model_of(self, formulas: AbstractSet[Formula]) -> bool:
        """Checks if the current model is a model for the given formulas.

        Returns:
            ``True`` if each of the given formulas evaluates to true in the
            current model for any assignment of elements from the universe of
            the current model to the free occurrences of variables in that
            formula, ``False`` otherwise.
        """
        for formula in formulas:
            assert formula.constants().issubset(self.constant_meanings.keys())
            for function,arity in formula.functions():
                assert function in self.function_meanings and \
                       self.function_arities[function] == arity
            for relation,arity in formula.relations():
                assert relation in self.relation_meanings and \
                       self.relation_arities[relation] in {-1, arity}
# Task 7.9

```