Contents

1 Basic Test Results

```
unzipping /tmp/bodek.lPpMXx/logic/ex7/yairgueta/presubmission/submission
Archive:  /tmp/bodek.lPpMXx/logic/ex7/yairgueta/presubmission/submission
 extracting: README
   creating: code/
   creating: code/predicates/
  inflating: code/predicates/semantics.py
  inflating: code/predicates/syntax.py
   creating: code/propositions/
  inflating: code/propositions/syntax.py

required files:
copying code/propositions/syntax.py
copying code/predicates/syntax.py
copying code/predicates/semantics.py

optional files:

README content:
    cs login 1: yairgueta
    cs login 2: noimimran



test_task1 Passed
test_task2 Passed
test_task3 Passed
test_task4 Passed
test_task5 Passed
test_task6 Passed
test_task7 Passed
test_task8 Passed
test_task9 Passed
```

2 README

```
yairgueta
noimimran
```

3 code/predicates/semantics.py

```python
# This file is part of the materials accompanying the book
# "Mathematical Logic through Python" by Gonczarowski and Nisan,
# Cambridge University Press. Book site: www.LogicThruPython.org
# (c) Yannai A. Gonczarowski and Noam Nisan, 2017-2020
# File name: predicates/semantics.py

"""Semantic analysis of predicate-logic expressions."""

from typing import AbstractSet, FrozenSet, Generic, Mapping, Tuple, TypeVar

from logic_utils import frozen, frozendict

from predicates.syntax import *

from itertools import product
#: 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 (`~typing.FrozenSet`\\[`T`]): the set of elements to which
            terms can be evaluated and over which quantifications are defined.
        constant_meanings (`~typing.Mapping`\\[`str`, `T`]): mapping from each
            constant name to the universe element to which it evaluates.
        relation_arities (`~typing.Mapping`\\[`str`, `int`]): mapping from
            each relation name to the arity of the relation, or to ``-1`` if the
            relation is the empty relation.
        relation_meanings (`~typing.Mapping`\\[`str`, `~typing.AbstractSet`\\[`~typing.Tuple`\\[`T`, ...]]]):
            mapping from each n-ary relation name to argument n-tuples (of
            universe elements) for which the relation is true.
        function_arities (`~typing.Mapping`\\[`str`, `int`]): mapping from
            each function name to the arity of the function.
        function_meanings (`~typing.Mapping`\\[`str`, `~typing.Mapping`\\[`~typing.Tuple`\\[`T`, ...], `T`]]):
            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_arities: Mapping[str, int]
    relation_meanings: Mapping[str, AbstractSet[Tuple[T, ...]]]
    function_arities: 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()):
        """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, "argument is: " + argument
            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)

    def __repr__(self) -> str:
        """Computes a string representation of the current model.

        Returns:
            A string representation of the current model.
        """
        return 'Universe=' + str(self.universe) + '; Constant Meanings=' + \
               str(self.constant_meanings) + '; Relation Meanings=' + \
               str(self.relation_meanings) + \
               ('; Function Meanings=' + str(self.function_meanings)
                if len(self.function_meanings) > 0 else '')

    def evaluate_term(self, term: Term,
```

```python
                      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
        if is_constant(term.root):
            return self.constant_meanings[term.root]
        if is_variable(term.root):
            return assignment[term.root]
        if is_function(term.root):
            # n = self.function_arities[term.root]
            args = tuple(self.evaluate_term(t, assignment) for t in
                         term.arguments)
            return self.function_meanings[term.root][args]

        # Task 7.7

    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
        if is_equality(formula.root):
            return self.evaluate_term(formula.arguments[0], assignment) == \
                   self.evaluate_term(formula.arguments[1], assignment)
        elif is_relation(formula.root):
            eval_terms = tuple(self.evaluate_term(t, assignment) for t in
                               formula.arguments)
            return eval_terms in self.relation_meanings[formula.root]
        elif is_unary(formula.root):
            return not self.evaluate_formula(formula.first, assignment)
```

```python
        elif is_binary(formula.root):
            p = self.evaluate_formula(formula.first, assignment)
            q = self.evaluate_formula(formula.second, assignment)
            if formula.root == '&':
                return p and q
            elif formula.root == '|':
                return p or q
            elif formula.root == '->':
                return (not p) or q
        else:
            if formula.root == 'A':
                for t in self.universe:
                    if not self.evaluate_formula(formula.predicate,
                                                 {**assignment, formula.variable: t}):
                        return False
                return True
            # quantifier
            else:
                for t in self.universe:
                    if self.evaluate_formula(formula.predicate,
                                             {**assignment, formula.variable: t}):
                        return True
                return False

    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
        free_vars = set()
        for f in formulas:
            free_vars.update(f.free_variables())
        for assignment in product(self.universe, repeat=len(free_vars)):
            params = dict(zip(free_vars, assignment))
            for f in formulas:
                if not self.evaluate_formula(f, params):
                    return False
        return True
```

4 code/predicates/syntax.py

```python
# This file is part of the materials accompanying the book
# "Mathematical Logic through Python" by Gonczarowski and Nisan,
# Cambridge University Press. Book site: www.LogicThruPython.org
# (c) Yannai A. Gonczarowski and Noam Nisan, 2017-2020
# File name: predicates/syntax.py

"""Syntactic handling of predicate-logic expressions."""

from __future__ import annotations
from typing import AbstractSet, Mapping, Optional, Sequence, Set, Tuple, Union

from logic_utils import fresh_variable_name_generator, frozen, \
    memoized_parameterless_method

from propositions.syntax import Formula as PropositionalFormula, \
    is_variable as is_propositional_variable
from functools import lru_cache


class ForbiddenVariableError(Exception):
    """Raised by `Term.substitute` and `Formula.substitute` when a substituted
    term contains a variable name that is forbidden in that context.

    Attributes:
        variable_name (`str`): the variable name that was forbidden in the
            context in which a term containing it was to be substituted.
    """
    variable_name: str

    def __init__(self, variable_name: str):
        """Initializes a `ForbiddenVariableError` from the offending variable
        name.

        Parameters:
            variable_name: variable name that is forbidden in the context in
                which a term containing it is to be substituted.
        """
        assert is_variable(variable_name)
        self.variable_name = variable_name


@lru_cache(maxsize=100)  # Cache the return value of is_constant
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] <= 'd')) and \
            string.isalnum()) or string == '_'


@lru_cache(maxsize=100)  # Cache the return value of is_variable
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()


@lru_cache(maxsize=100)  # Cache the return value of is_function
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()


def find_relevant_part(string: str, func):
    i = 1
    while i < len(string) + 1:
        if func(string[0:i]):
            i += 1
        else:
            i -= 1
            break
    return i


@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 (`str`): the constant name, variable name, or function name at the
            root of the term tree.
        arguments (`~typing.Optional`\\[`~typing.Tuple`\\[`Term`, ...]]): 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):
        """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

    @memoized_parameterless_method
    def __repr__(self) -> str:
```

```python
        """Computes the string representation of the current term.

        Returns:
            The standard string representation of the current term.
        """
        # Task 7.1
        if not is_function(self.root):
            return self.root
        else:
            return self.root + "(" + ",".join([str(x) for x in
                                               self.arguments]) + ")"

    def __eq__(self, other: object) -> bool:
        """Compares the current term with the given one.

        Parameters:
            other: object to compare to.

        Returns:
            ``True`` if the given object is a `Term` object that equals the
            current term, ``False`` otherwise.
        """
        return isinstance(other, Term) and str(self) == str(other)

    def __ne__(self, other: object) -> bool:
        """Compares the current term with the given one.

        Parameters:
            other: object to compare to.

        Returns:
            ``True`` if the given object is not a `Term` object or does not
            equal the current term, ``False`` otherwise.
        """
        return not self == other

    def __hash__(self) -> int:
        return hash(str(self))

    @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 7.3.1
        if is_variable(string[0]) or is_constant(string[0]):
            i = find_relevant_part(string, lambda s: is_constant(s) or
                                                      is_variable(s))

            return Term(string[:i]), string[i:]
        elif is_function(string[0]):
            terms_lst = []
            i = string.index('(')
            t, rest = Term._parse_prefix(string[i + 1:])
            terms_lst.append(t)
            while rest[0] == ',':
                t, rest = Term._parse_prefix(rest[1:])
                terms_lst.append(t)
            return Term(string[:i], terms_lst), rest[1:]
```

```python

    @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 7.3.2
        prefix, suffix = Term._parse_prefix(string)
        assert prefix is not None and len(suffix) == 0
        return prefix

    def __collect_vars(self, final_set, func):
        if is_function(self.root):
            if func(self.root):
                final_set.add((self.root, len(self.arguments)))
            for arg in self.arguments:
                arg.__collect_vars(final_set, func)
        elif func(self.root):
            final_set.add(self.root)
        else:
            return

    @memoized_parameterless_method
    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
        final_set = set()
        self.__collect_vars(final_set, is_constant)
        return final_set

    @memoized_parameterless_method
    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
        final_set = set()
        self.__collect_vars(final_set, is_variable)
        return final_set

    @memoized_parameterless_method
    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
        final_set = set()
        self.__collect_vars(final_set, is_function)
        return final_set

    def substitute(self, substitution_map: Mapping[str, Term],
                   forbidden_variables: AbstractSet[str] = frozenset()) -> Term:
        """Substitutes in the current term, each constant name `name` or
```

```
        variable name `name` that is a key in `substitution_map` with the term
        `substitution_map`\ ``[``\ `name`\ ``]``.

        Parameters:
            substitution_map: mapping defining the substitutions to be
                performed.
            forbidden_variables: variables not allowed in substitution terms.

        Returns:
            The term resulting from performing all substitutions. Only
            constant names and variable names originating in the current term
            are substituted (i.e., those originating in one of the specified
            substitutions are not subjected to additional substitutions).

        Raises:
            ForbiddenVariableError: If a term that is used in the requested
                substitution contains a variable from `forbidden_variables`.

        Examples:
            >>> Term.parse('f(x,c)').substitute(
            ...     {'c': Term.parse('plus(d,x)'), 'x': Term.parse('c')}, {'y'})
            f(c,plus(d,x))

            >>> Term.parse('f(x,c)').substitute(
            ...     {'c': Term.parse('plus(d,y)')}, {'y'})
            Traceback (most recent call last):
              ...
            predicates.syntax.ForbiddenVariableError: y
        """
        for element_name in substitution_map:
            assert is_constant(element_name) or is_variable(element_name)
        for variable in forbidden_variables:
            assert is_variable(variable)
        # Task 9.1
        for val in substitution_map.values():
            intersection = val.variables().intersection(forbidden_variables)
            if intersection:
                raise ForbiddenVariableError(next(iter(intersection)))
        return self.__substitute_helper(substitution_map)

    def __substitute_helper(self, substitution_map: Mapping[str, Term]) -> Term:
        if is_constant(self.root) or is_variable(self.root):
            temp = substitution_map.get(self.root)
            if temp is not None:
                return temp
            return self
        else:
            return Term(self.root,
                        [s.__substitute_helper(substitution_map) for
                         s in self.arguments])


@lru_cache(maxsize=100)  # Cache the return value of is_equality
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.
    """
    return string == '='


@lru_cache(maxsize=100)  # Cache the return value of is_relation
def is_relation(string: str) -> bool:
```

```python
    """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()


@lru_cache(maxsize=100)  # Cache the return value of is_unary
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 == '~'


@lru_cache(maxsize=100)  # Cache the return value of is_binary
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 == '->'


@lru_cache(maxsize=100)  # Cache the return value of is_quantifier
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 (`str`): the relation name, equality relation, operator, or
            quantifier at the root of the formula tree.
        arguments (`~typing.Optional`\\[`~typing.Tuple`\\[`Term`, ...]]): the
            arguments to the root, if the root is a relation name or the
            equality relation.
        first (`~typing.Optional`\\[`Formula`]): the first operand to the root,
            if the root is a unary or binary operator.
        second (`~typing.Optional`\\[`Formula`]): the second
            operand to the root, if the root is a binary operator.
        variable (`~typing.Optional`\\[`str`]): the variable name quantified by
            the root, if the root is a quantification.
```

```
        predicate (`~typing.Optional`\\[`Formula`]): 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):
        """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 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

    @memoized_parameterless_method
    def __repr__(self) -> str:
        """Computes the string representation of the current formula.

        Returns:
            The standard string representation of the current formula.
        """
        # Task 7.2
        if is_equality(self.root):
            return str(self.arguments[0]) + "=" + str(self.arguments[1])
        elif is_relation(self.root):
```

```
            return self.root + "(" + ",".join([str(x) for x in
                                               self.arguments]) + ")"
        elif is_unary(self.root):
            return self.root + str(self.first)
        elif is_binary(self.root):
            return "(" + str(self.first) + self.root + str(self.second) + ")"
        else:
            # quantifier
            return self.root + self.variable + "[" + str(self.predicate) + "]"

    def __eq__(self, other: object) -> bool:
        """Compares the current formula with the given one.

        Parameters:
            other: object to compare to.

        Returns:
            ``True`` if the given object is a `Formula` object that equals the
            current formula, ``False`` otherwise.
        """
        return isinstance(other, Formula) and str(self) == str(other)

    def __ne__(self, other: object) -> bool:
        """Compares the current formula with the given one.

        Parameters:
            other: object to compare to.

        Returns:
            ``True`` if the given object is not a `Formula` object or does not
            equal the current formula, ``False`` otherwise.
        """
        return not self == other

    def __hash__(self) -> int:
        return hash(str(self))

    @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., ``'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

        if is_relation(string[0]):
            terms = []
            i = string.index('(')
            if string[i + 1] != ')':
                t, rest = Term._parse_prefix(string[i + 1:])
                terms.append(t)
                while rest[0] == ',':
                    t, rest = Term._parse_prefix(rest[1:])
                    terms.append(t)
                rest = rest[1:]
            else:
                rest = string[i + 2:]
            return Formula(string[:i], terms), rest
        elif is_unary(string[0]):
```

```python
            f, rest = Formula._parse_prefix(string[1:])
            return Formula('~', f), rest
        elif string[0] == '(':
            first, rest = Formula._parse_prefix(string[1:])
            if is_binary(rest[0]):
                i = 1
            else:
                i = 2
            operator, rest = rest[:i], rest[i:]
            second, rest2 = Formula._parse_prefix(rest)
            return Formula(operator, first, second), rest2[1:]
        elif is_quantifier(string[0]):
            i = string.index('[')
            var_name = string[1:i]
            f, rest = Formula._parse_prefix(string[i + 1:])
            return Formula(string[0], var_name, f), rest[1:]
        else:
            # i = string.index('=')
            t1, rest = Term._parse_prefix(string)
            t2, rest = Term._parse_prefix(rest[1:])
            return Formula('=', [t1, t2]), rest

    @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.
        """
        parsed, rest = Formula._parse_prefix(string)
        assert rest is not None
        return parsed

    @memoized_parameterless_method
    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
        if is_equality(self.root) or is_relation(self.root):
            s = set()
            for term in self.arguments:
                s.update(term.constants())
            return s
        elif is_unary(self.root):
            return self.first.constants()
        elif is_binary(self.root):
            return self.first.constants().union(self.second.constants())
        else:
            # quantifier
            return self.predicate.constants()

    @memoized_parameterless_method
    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
        if is_equality(self.root) or is_relation(self.root):
            s = set()
            for term in self.arguments:
```

```
                s.update(term.variables())
            return s
        elif is_unary(self.root):
            return self.first.variables()
        elif is_binary(self.root):
            return self.first.variables().union(self.second.variables())
        else:
            # quantifier
            return self.predicate.variables().union({self.variable})

    @memoized_parameterless_method
    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 7.6.3
        if is_equality(self.root) or is_relation(self.root):
            s = set()
            for term in self.arguments:
                s.update(term.variables())
            return s
        elif is_unary(self.root):
            return self.first.free_variables()
        elif is_binary(self.root):
            return self.first.free_variables().union(
                self.second.free_variables())
        else:
            # quantifier
            x = self.predicate.free_variables()
            x.discard(self.variable)
            return x

    @memoized_parameterless_method
    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
        if is_equality(self.root) or is_relation(self.root):
            s = set()
            for term in self.arguments:
                s.update(term.functions())
            return s
        elif is_unary(self.root):
            return self.first.functions()
        elif is_binary(self.root):
            return self.first.functions().union(self.second.functions())
        else:
            # quantifier
            return self.predicate.functions()

    @memoized_parameterless_method
    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
```

```python
        if is_equality(self.root):
            return set()
        elif is_relation(self.root):
            return {(self.root, len(self.arguments))}
        elif is_unary(self.root):
            return self.first.relations()
        elif is_binary(self.root):
            return self.first.relations().union(self.second.relations())
        else:
            # quantifier
            return self.predicate.relations()

    def substitute(self, substitution_map: Mapping[str, Term],
                   forbidden_variables: AbstractSet[str] = frozenset()) -> \
            Formula:
        """Substitutes in the current formula, each constant name `name` or free
        occurrence of variable name `name` that is a key in `substitution_map`
        with the term `substitution_map`\ ``[``\ `name`\ ``]``.

        Parameters:
            substitution_map: mapping defining the substitutions to be
                performed.
            forbidden_variables: variables not allowed in substitution terms.

        Returns:
            The formula resulting from performing all substitutions. Only
            constant names and variable names originating in the current formula
            are substituted (i.e., those originating in one of the specified
            substitutions are not subjected to additional substitutions).

        Raises:
            ForbiddenVariableError: If a term that is used in the requested
                substitution contains a variable from `forbidden_variables`
                or a variable occurrence that becomes bound when that term is
                substituted into the current formula.

        Examples:
            >>> Formula.parse('Ay[x=c]').substitute(
            ...     {'c': Term.parse('plus(d,x)'), 'x': Term.parse('c')}, {'z'})
            Ay[c=plus(d,x)]

            >>> Formula.parse('Ay[x=c]').substitute(
            ...     {'c': Term.parse('plus(d,z)')}, {'z'})
            Traceback (most recent call last):
              ...
            predicates.syntax.ForbiddenVariableError: z

            >>> Formula.parse('Ay[x=c]').substitute(
            ...     {'c': Term.parse('plus(d,y)')})
            Traceback (most recent call last):
              ...
            predicates.syntax.ForbiddenVariableError: y
        """
        for element_name in substitution_map:
            assert is_constant(element_name) or is_variable(element_name)
        for variable in forbidden_variables:
            assert is_variable(variable)
        # Task 9.2
        return self.__substitute_formula_helper(forbidden_variables, substitution_map, self.free_variables())

    def __substitute_formula_helper(self, forbidden_variables, substitution_map, free_vars):
        if is_equality(self.root) or is_relation(self.root):
            new_arguments = []
            for arg in self.arguments:
                if not arg.variables().issubset(free_vars):
                    new_arguments.append(arg)
                else:
                    new_arguments.append(arg.substitute(substitution_map, forbidden_variables))
```

```
            return Formula(self.root, new_arguments)
        elif is_unary(self.root):
            return Formula(self.root,
                           self.first.substitute(substitution_map,
                                                 forbidden_variables))
        elif is_binary(self.root):
            return Formula(self.root,
                           self.first.substitute(substitution_map,
                                                 forbidden_variables),
                           self.second.substitute(substitution_map,
                                                  forbidden_variables))
        else:
            return Formula(self.root, self.variable,
                           self.predicate.substitute(substitution_map,
                                                     set(forbidden_variables).union({self.variable})))

    def propositional_skeleton(self) -> Tuple[PropositionalFormula,
                                              Mapping[str, Formula]]:
        """Computes a propositional skeleton of the current formula.

        Returns:
            A pair. The first element of the pair is a propositional formula
            obtained from the current formula by substituting every (outermost)
            subformula that has a relation or quantifier at its root with an
            atomic propositional formula, consistently such that multiple equal
            such (outermost) subformulas are substituted with the same atomic
            propositional formula. The atomic propositional formulas used for
            substitution are obtained, from left to right, by calling
            `next`\ ``(``\ `~logic_utils.fresh_variable_name_generator`\ ``)``.
            The second element of the pair is a mapping from each atomic
            propositional formula to the subformula for which it was
            substituted.

        Examples:
            >>> formula = Formula.parse('((Ax[x=7]&x=7)|(x=7->~Q(y)))')
            >>> formula.propositional_skeleton()
            (((z1&z2)|(z2->~z3)), {'z1': Ax[x=7], 'z2': x=7, 'z3': Q(y)})
            >>> formula.propositional_skeleton()
            (((z4&z5)|(z5->~z6)), {'z4': Ax[x=7], 'z5': x=7, 'z6': Q(y)})
        """
        # Task 9.8
        var_map = dict()
        return self.__skeleton_helper(var_map), {key: val for val, key in var_map.items()}

    def __skeleton_helper(self, var_map):
        if is_equality(self.root) or is_relation(self.root) or is_quantifier(self.root):
            zi = var_map.get(self)
            if not zi:
                zi = next(fresh_variable_name_generator)
                var_map.update({self: zi})
            return PropositionalFormula(zi)
        elif is_unary(self.root):
            return PropositionalFormula(self.root, self.first.__skeleton_helper(var_map))
        elif is_binary(self.root):
            return PropositionalFormula(self.root, self.first.__skeleton_helper(var_map),
                                        self.second.__skeleton_helper(var_map))

    @staticmethod
    def from_propositional_skeleton(skeleton: PropositionalFormula,
                                    substitution_map: Mapping[str, Formula]) -> \
            Formula:
        """Computes a predicate-logic formula from a propositional skeleton and
        a substitution map.

        Arguments:
            skeleton: propositional skeleton for the formula to compute,
                containing no constants or operators beyond ``'~'``, ``'->'``,
                ``'|'``, and ``'&'``.
```

```
            substitution_map: mapping from each atomic propositional subformula
                of the given skeleton to a predicate-logic formula.

        Returns:
            A predicate-logic formula obtained from the given propositional
            skeleton by substituting each atomic propositional subformula with
            the formula mapped to it by the given map.

        Examples:
            >>> Formula.from_propositional_skeleton(
            ...     PropositionalFormula.parse('((z1&z2)|(z2->~z3))'),
            ...     {'z1': Formula.parse('Ax[x=7]'), 'z2': Formula.parse('x=7'),
            ...      'z3': Formula.parse('Q(y)')})
            ((Ax[x=7]&x=7)|(x=7->~Q(y)))
        """
        for operator in skeleton.operators():
            assert is_unary(operator) or is_binary(operator)
        for variable in skeleton.variables():
            assert variable in substitution_map
        # Task 9.10
        if is_propositional_variable(skeleton.root):
            return substitution_map[skeleton.root]
        elif is_unary(skeleton.root):
            return Formula(skeleton.root, Formula.from_propositional_skeleton(skeleton.first, substitution_map))
        else:
            return Formula(skeleton.root, Formula.from_propositional_skeleton(skeleton.first,
                                                                              substitution_map),
                           Formula.from_propositional_skeleton(skeleton.second, substitution_map))
```

5 code/propositions/syntax.py

```python
# This file is part of the materials accompanying the book
# "Mathematical Logic through Python" by Gonczarowski and Nisan,
# Cambridge University Press. Book site: www.LogicThruPython.org
# (c) Yannai A. Gonczarowski and Noam Nisan, 2017-2020
# File name: propositions/syntax.py

"""Syntactic handling of propositional formulas."""

from __future__ import annotations
from functools import lru_cache
from typing import Mapping, Optional, Set, Tuple, Union

from logic_utils import frozen, memoized_parameterless_method


@lru_cache(maxsize=100)  # Cache the return value of is_variable
def is_variable(string: str) -> bool:
    """Checks if the given string is an atomic proposition.

    Parameters:
        string: string to check.

    Returns:
        ``True`` if the given string is an atomic proposition, ``False``
        otherwise.
    """
    return string[0] >= 'p' and string[0] <= 'z' and \
           (len(string) == 1 or string[1:].isdigit())


@lru_cache(maxsize=100)  # Cache the return value of is_constant
def is_constant(string: str) -> bool:
    """Checks if the given string is a constant.

    Parameters:
        string: string to check.

    Returns:
        ``True`` if the given string is a constant, ``False`` otherwise.
    """
    return string == 'T' or string == 'F'


@lru_cache(maxsize=100)  # Cache the return value of is_unary
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 == '~'


@lru_cache(maxsize=100)  # Cache the return value of is_binary
def is_binary(string: str) -> bool:
    """Checks if the given string is a binary operator.
```

```python

    Parameters:
        string: string to check.

    Returns:
        ``True`` if the given string is a binary operator, ``False`` otherwise.
    """
    # return string == '&' or string == '|' or string == '->'
    # For Chapter 3:
    return string in {'&', '|', '->', '+', '<->', '-&', '-|'}


@frozen
class Formula:
    """An immutable propositional formula in tree representation, composed from
    atomic propositions, and operators applied to them.

    Attributes:
        root (`str`): the constant, atomic proposition, or operator at the root
            of the formula tree.
        first (`~typing.Optional`\\[`Formula`]): the first operand to the root,
            if the root is a unary or binary operator.
        second (`~typing.Optional`\\[`Formula`]): the second operand to the
            root, if the root is a binary operator.
    """
    root: str
    first: Optional[Formula]
    second: Optional[Formula]

    def __init__(self, root: str, first: Optional[Formula] = None,
                 second: Optional[Formula] = None):
        """Initializes a `Formula` from its root and root operands.

        Parameters:
            root: the root for the formula tree.
            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.
        """
        if is_variable(root) or is_constant(root):
            assert first is None and second is None
            self.root = root
        elif is_unary(root):
            assert first is not None and second is None
            self.root, self.first = root, first
        else:
            assert is_binary(root)
            assert first is not None and second is not None
            self.root, self.first, self.second = root, first, second

    @memoized_parameterless_method
    def __repr__(self) -> str:
        """Computes the string representation of the current formula.

        Returns:
            The standard string representation of the current formula.
        """
        if is_variable(self.root) or is_constant(self.root):
            return self.root
        elif is_unary(self.root):
            return self.root + str(self.first)
        else:
            return "(" + str(self.first) + self.root + str(self.second) + ")"

    def __eq__(self, other: object) -> bool:
        """Compares the current formula with the given one.

```

```
        Parameters:
            other: object to compare to.

        Returns:
            ``True`` if the given object is a `Formula` object that equals the
            current formula, ``False`` otherwise.
        """
        return isinstance(other, Formula) and str(self) == str(other)

    def __ne__(self, other: object) -> bool:
        """Compares the current formula with the given one.

        Parameters:
            other: object to compare to.

        Returns:
            ``True`` if the given object is not a `Formula` object or does not
            equal the current formula, ``False`` otherwise.
        """
        return not self == other

    def __hash__(self) -> int:
        return hash(str(self))

    @memoized_parameterless_method
    def variables(self) -> Set[str]:
        """Finds all atomic propositions (variables) in the current formula.

        Returns:
            A set of all atomic propositions used in the current formula.
        """
        if is_variable(self.root):
            return {self.root}
        elif is_constant(self.root):
            return set()
        elif is_unary(self.root):
            return self.first.variables()
        else:
            return self.first.variables().union(self.second.variables())

    @memoized_parameterless_method
    def operators(self) -> Set[str]:
        """Finds all operators in the current formula.

        Returns:
            A set of all operators (including ``'T'`` and ``'F'``) used in the
            current formula.
        """
        if is_variable(self.root):
            return set()
        elif is_constant(self.root):
            return {self.root}
        elif is_unary(self.root):
            return {self.root}.union(self.first.operators())
        else:
            return {self.root}.union(self.first.operators(), self.second.operators())

    @staticmethod
    def _parse_prefix(string: str) -> Tuple[Union[Formula, None], str]:
        """Parses a prefix of the given string into a formula.

        Parameters:
            string: string to parse.

        Returns:
            A pair of the parsed formula and the unparsed suffix of the string.
            If the given string has as a prefix a variable name (e.g.,
            ``'x12'``) or a unary operator follows by a variable name, then the
```

```
            parsed prefix will include that entire variable name (and not just a
            part of it, such as ``'x1'``). If no prefix of the given string is a
            valid standard string representation of a formula then returned pair
            should be of ``None`` and an error message, where the error message
            is a string with some human-readable content.
        """
        if len(string) == 0:
            return None, "Formula is empty."
        elif is_constant(string[0]):
            return Formula(string[0]), string[1:]
        elif is_variable(string[0]):
            i = 1
            while i < len(string) + 1:
                if is_variable(string[0:i]):
                    i += 1
                else:
                    i -= 1
                    break
            return Formula(string[0:i]), string[i:]
        elif string[0] == "(":
            first, rest = Formula._parse_prefix(string[1:])
            operator = ""
            rester = rest
            i = 0
            while (not is_binary(operator)) and i < len(rester):
                operator = rester[0:i]
                rest = rester[i:]
                i = i + 1

            if not is_binary(operator):
                return None, "Binary operators must be one of: &, |, ->"

            second, rest2 = Formula._parse_prefix(rest)
            if first is None or second is None or len(rest2) == 0 or rest2[0] != ")":
                return None, "The use of binary operator is: '(<valid formula1>*<valid formula2>)', where * is" \
                             " the binary operator."

            return Formula(operator, first, second), rest2[1:]
        elif is_unary(string[0]):
            f, rest = Formula._parse_prefix(string[1:])
            if f is None:
                return None, "The use of not operator is: '~<valid formula>'"
            return Formula(string[0], f), rest
        else:
            return None, "Expected valid formula."

    @staticmethod
    def is_formula(string: str) -> bool:
        """Checks if the given string is a valid representation of a formula.

        Parameters:
            string: string to check.

        Returns:
            ``True`` if the given string is a valid standard string
            representation of a formula, ``False`` otherwise.
        """
        prefix, suffix = Formula._parse_prefix(string)
        return prefix is not None and len(suffix) == 0

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

        Parameters:
            string: string to parse.

        Returns:
```

```python
            A formula whose standard string representation is the given string.
        """
        prefix, suffix = Formula._parse_prefix(string)
        assert prefix is not None and len(suffix) == 0
        return prefix

    # Optional tasks for Chapter 1

    def polish(self) -> str:
        """Computes the polish notation representation of the current formula.

        Returns:
            The polish notation representation of the current formula.
        """
        if is_variable(self.root) or is_constant(self.root):
            return self.root
        elif is_unary(self.root):
            return self.root + self.first.polish()
        else:
            return self.root + self.first.polish() + self.second.polish()

    @staticmethod
    def _parse_polish_prefix(string: str) -> Tuple[Union[Formula, None], str]:
        if is_constant(string[0]):
            return Formula(string[0]), string[1:]
        elif is_variable(string[0]):
            i = 1
            while i < len(string) + 1:
                if is_variable(string[0:i]):
                    i += 1
                else:
                    i -= 1
                    break
            return Formula(string[0:i]), string[i:]
        elif is_unary(string[0]):
            f, rest = Formula._parse_polish_prefix(string[1:])
            if f is None:
                return None, "The use of not operator is: '~<valid formula>'"
            return Formula(string[0], f), rest
        else:
            if is_binary(string[0]):
                operator = string[0]
                string = string[1:]
            elif is_binary(string[0:2]):
                operator = string[0:2]
                string = string[2:]
            else:
                return None, "Expected valid formula."
            first, rest1 = Formula._parse_polish_prefix(string)
            second, rest2 = Formula._parse_polish_prefix(rest1)
            if first is None or second is None:
                return None, "The use of binary operator is: *<f1><f2>"
            return Formula(operator, first, second), rest2

    @staticmethod
    def parse_polish(string: str) -> Formula:
        """Parses the given polish notation representation into a formula.

        Parameters:
            string: string to parse.

        Returns:
            A formula whose polish notation representation is the given string.
        """
        return Formula._parse_polish_prefix(string)[0]

    def substitute_variables(self, substitution_map: Mapping[str, Formula]) -> \
            Formula:
```

```python
        """Substitutes in the current formula, each variable `v` that is a key
        in `substitution_map` with the formula `substitution_map[v]`.

        Parameters:
            substitution_map: mapping defining the substitutions to be
                performed.

        Returns:
            The formula resulting from performing all substitutions. Only
            variables originating in the current formula are substituted (i.e.,
            variables originating in one of the specified substitutions are not
            subjected to additional substitutions).

        Examples:
            >>> Formula.parse('((p->p)|r)').substitute_variables(
            ...     {'p': Formula.parse('(q&r)'), 'r': Formula.parse('p')})
            (((q&r)->(q&r))|p)
        """
        for variable in substitution_map:
            assert is_variable(variable)
        if is_variable(self.root) or is_constant(self.root):
            return substitution_map.get(self.root, self)
        if is_unary(self.root):
            return Formula(self.root, self.first.substitute_variables(substitution_map))
        if is_binary(self.root):
            return Formula(self.root, self.first.substitute_variables(substitution_map),
                           self.second.substitute_variables(substitution_map))

    def substitute_operators(self, substitution_map: Mapping[str, Formula]) -> \
            Formula:
        """Substitutes in the current formula, each constant or operator `op`
        that is a key in `substitution_map` with the formula
        `substitution_map[op]` applied to its (zero or one or two) operands,
        where the first operand is used for every occurrence of ``'p'`` in the
        formula and the second for every occurrence of ``'q'``.

        Parameters:
            substitution_map: mapping defining the substitutions to be
                performed.

        Returns:
            The formula resulting from performing all substitutions. Only
            operators originating in the current formula are substituted (i.e.,
            operators originating in one of the specified substitutions are not
            subjected to additional substitutions).

        Examples:
            >>> Formula.parse('((x&y)&~z)').substitute_operators(
            ...     {'&': Formula.parse('~(~p|~q)')})
            ~(~~(~x|~y)|~~z)
        """
        for operator in substitution_map:
            assert is_binary(operator) or is_unary(operator) or \
                   is_constant(operator)
            assert substitution_map[operator].variables().issubset({'p', 'q'})

        p = Formula("p")
        q = Formula("q")
        if is_variable(self.root):
            return self
        if is_constant(self.root):
            return substitution_map.get(self.root, self)
        if is_unary(self.root):
            _map = {'p': self.first.substitute_operators(substitution_map)}
            return substitution_map.get(self.root, Formula(self.root, p)).substitute_variables(_map)
        if is_binary(self.root):
            _map = {'p': self.first.substitute_operators(substitution_map),
                    'q': self.second.substitute_operators(substitution_map)}
```

```
        return substitution_map.get(self.root, Formula(self.root, p, q)).substitute_variables(_map)
```