

# Contents

<b>1</b>	<b>Basic Test Results</b>	<b>2</b>
<b>2</b>	<b>README</b>	<b>3</b>
<b>3</b>	<b>code/predicates/semantics.py</b>	<b>4</b>
<b>4</b>	<b>code/predicates/syntax.py</b>	<b>8</b>
<b>5</b>	<b>code/propositions/syntax.py</b>	<b>21</b>

# 1 Basic Test Results

```
1  unzipping /tmp/bodek.lPpMXx/logic/ex7/yairgueta/presubmission/submission
2  Archive:  /tmp/bodek.lPpMXx/logic/ex7/yairgueta/presubmission/submission
3  extracting: README
4      creating: code/
5      creating: code/predicates/
6      inflating: code/predicates/semantics.py
7      inflating: code/predicates/syntax.py
8      creating: code/propositions/
9      inflating: code/propositions/syntax.py
10
11  required files:
12  copying code/propositions/syntax.py
13  copying code/predicates/syntax.py
14  copying code/predicates/semantics.py
15
16  optional files:
17
18  README content:
19      cs login 1: yairgueta
20      cs login 2: noimimran
21
22
23
24  test_task1 Passed
25  test_task2 Passed
26  test_task3 Passed
27  test_task4 Passed
28  test_task5 Passed
29  test_task6 Passed
30  test_task7 Passed
31  test_task8 Passed
32  test_task9 Passed
```

## 2 README

```
1 yairgueta
2 noimimran
```

## 3 code/predicates/semantics.py

```
1  # This file is part of the materials accompanying the book
2  # "Mathematical Logic through Python" by Gonczarowski and Nisan,
3  # Cambridge University Press. Book site: www.LogicThruPython.org
4  # (c) Yannai A. Gonczarowski and Noam Nisan, 2017-2020
5  # File name: predicates/semantics.py
6
7  """Semantic analysis of predicate-logic expressions."""
8
9  from typing import AbstractSet, FrozenSet, Generic, Mapping, Tuple, TypeVar
10
11  from logic_utils import frozen, frozendict
12
13  from predicates.syntax import *
14
15  from itertools import product
16  #: A generic type for a universe element in a model.
17  T = TypeVar('T')
18
19  @frozen
20  class Model(Generic[T]):
21      """An immutable model for predicate-logic constructs.
22
23      Attributes:
24          universe (~typing.FrozenSet`\\[T`]): the set of elements to which
25          terms can be evaluated and over which quantifications are defined.
26          constant_meanings (~typing.Mapping`\\[str`, T]): mapping from each
27          constant name to the universe element to which it evaluates.
28          relation_arities (~typing.Mapping`\\[str`, int]): mapping from
29          each relation name to the arity of the relation, or to -1 if the
30          relation is the empty relation.
31          relation_meanings (~typing.Mapping`\\[str`, ~typing.AbstractSet`\\[~typing.Tuple`\\[T, ...]]]):
32          mapping from each n-ary relation name to argument n-tuples (of
33          universe elements) for which the relation is true.
34          function_arities (~typing.Mapping`\\[str`, int]): mapping from
35          each function name to the arity of the function.
36          function_meanings (~typing.Mapping`\\[str`, ~typing.Mapping`\\[~typing.Tuple`\\[T, ...], T]):
37          mapping from each n-ary function name to the mapping from each
38          argument n-tuple (of universe elements) to the universe element that
39          the function outputs given these arguments.
40      """
41      universe: FrozenSet[T]
42      constant_meanings: Mapping[str, T]
43      relation_arities: Mapping[str, int]
44      relation_meanings: Mapping[str, AbstractSet[Tuple[T, ...]]]
45      function_arities: Mapping[str, int]
46      function_meanings: Mapping[str, Mapping[Tuple[T, ...], T]]
47
48      def __init__(self, universe: AbstractSet[T],
49                  constant_meanings: Mapping[str, T],
50                  relation_meanings: Mapping[str, AbstractSet[Tuple[T, ...]]],
51                  function_meanings: Mapping[str, Mapping[Tuple[T, ...], T]] =
52                      frozendict()):
53          """Initializes a Model from its universe and constant, relation, and
54          function meanings.
55
56          Parameters:
57              universe: the set of elements to which terms are to be evaluated
58              and over which quantifications are to be defined.
59              constant_meanings: mapping from each constant name to a universe
```

```

60         element to which it is to be evaluated.
61     relation_meanings: mapping from each relation name that is to
62         be the name of an n-ary relation, to the argument n-tuples (of
63         universe elements) for which the relation is to be true.
64     function_meanings: mapping from each function name that is to
65         be the name of an n-ary function, to a mapping from each
66         argument n-tuple (of universe elements) to a universe element
67         that the function is to output given these arguments.
68 """
69 self.universe = frozenset(universe)
70
71 for constant in constant_meanings:
72     assert is_constant(constant)
73     assert constant_meanings[constant] in universe
74 self.constant_meanings = frozendict(constant_meanings)
75
76 relation_aritys = {}
77 for relation in relation_meanings:
78     assert is_relation(relation)
79     relation_meaning = relation_meanings[relation]
80     if len(relation_meaning) == 0:
81         arity = -1 # any
82     else:
83         some_arguments = next(iter(relation_meaning))
84         arity = len(some_arguments)
85         for arguments in relation_meaning:
86             assert len(arguments) == arity
87             for argument in arguments:
88                 assert argument in universe, "argument is: " + argument
89     relation_aritys[relation] = arity
90 self.relation_meanings = \
91     frozendict({relation: frozenset(relation_meanings[relation]) for
92         relation in relation_meanings})
93 self.relation_aritys = frozendict(relation_aritys)
94
95 function_aritys = {}
96 for function in function_meanings:
97     assert is_function(function)
98     function_meaning = function_meanings[function]
99     assert len(function_meaning) > 0
100     some_argument = next(iter(function_meaning))
101     arity = len(some_argument)
102     assert arity > 0
103     assert len(function_meaning) == len(universe)**arity
104     for arguments in function_meaning:
105         assert len(arguments) == arity
106         for argument in arguments:
107             assert argument in universe
108             assert function_meaning[arguments] in universe
109     function_aritys[function] = arity
110 self.function_meanings = \
111     frozendict({function: frozendict(function_meanings[function]) for
112         function in function_meanings})
113 self.function_aritys = frozendict(function_aritys)
114
115 def __repr__(self) -> str:
116     """Computes a string representation of the current model.
117
118     Returns:
119         A string representation of the current model.
120     """
121     return 'Universe=' + str(self.universe) + '; Constant Meanings=' + \
122         str(self.constant_meanings) + '; Relation Meanings=' + \
123         str(self.relation_meanings) + \
124         ('; Function Meanings=' + str(self.function_meanings)
125         if len(self.function_meanings) > 0 else '')
126
127 def evaluate_term(self, term: Term,

```

```

128         assignment: Mapping[str, T] = frozendict()) -> T:
129         """Calculates the value of the given term in the current model, for the
130 given assignment of values to variables names.
131
132     Parameters:
133         term: term to calculate the value of, for the constants and
134             functions of which the current model has meanings.
135         assignment: mapping from each variable name in the given term to a
136             universe element to which it is to be evaluated.
137
138     Returns:
139         The value (in the universe of the current model) of the given
140         term in the current model, for the given assignment of values to
141         variable names.
142     """
143     assert term.constants().issubset(self.constant_meanings.keys())
144     assert term.variables().issubset(assignment.keys())
145     for function, arity in term.functions():
146         assert function in self.function_meanings and \
147             self.function_arities[function] == arity
148     if is_constant(term.root):
149         return self.constant_meanings[term.root]
150     if is_variable(term.root):
151         return assignment[term.root]
152     if is_function(term.root):
153         # n = self.function_arities[term.root]
154         args = tuple(self.evaluate_term(t, assignment) for t in
155                     term.arguments)
156         return self.function_meanings[term.root][args]
157
158     # Task 7.7
159
160 def evaluate_formula(self, formula: Formula,
161                     assignment: Mapping[str, T] = frozendict()) -> bool:
162     """Calculates the truth value of the given formula in the current model,
163 for the given assignment of values to free occurrences of variables
164 names.
165
166     Parameters:
167         formula: formula to calculate the truth value of, for the constants,
168             functions, and relations of which the current model has
169             meanings.
170         assignment: mapping from each variable name that has a free
171             occurrence in the given formula to a universe element to which
172             it is to be evaluated.
173
174     Returns:
175         The truth value of the given formula in the current model, for the
176         given assignment of values to free occurrences of variable names.
177     """
178     assert formula.constants().issubset(self.constant_meanings.keys())
179     assert formula.free_variables().issubset(assignment.keys())
180     for function, arity in formula.functions():
181         assert function in self.function_meanings and \
182             self.function_arities[function] == arity
183     for relation, arity in formula.relations():
184         assert relation in self.relation_meanings and \
185             self.relation_arities[relation] in {-1, arity}
186
187     # Task 7.8
188     if is_equality(formula.root):
189         return self.evaluate_term(formula.arguments[0], assignment) == \
190             self.evaluate_term(formula.arguments[1], assignment)
191     elif is_relation(formula.root):
192         eval_terms = tuple(self.evaluate_term(t, assignment) for t in
193                           formula.arguments)
194         return eval_terms in self.relation_meanings[formula.root]
195     elif is_unary(formula.root):
196         return not self.evaluate_formula(formula.first, assignment)

```

```

196     elif is_binary(formula.root):
197         p = self.evaluate_formula(formula.first, assignment)
198         q = self.evaluate_formula(formula.second, assignment)
199         if formula.root == '&':
200             return p and q
201         elif formula.root == '|':
202             return p or q
203         elif formula.root == '->':
204             return (not p) or q
205     else:
206         if formula.root == 'A':
207             for t in self.universe:
208                 if not self.evaluate_formula(formula.predicate,
209                                             {**assignment, formula.variable: t}):
210                     return False
211             return True
212         # quantifier
213     else:
214         for t in self.universe:
215             if self.evaluate_formula(formula.predicate,
216                                     {**assignment, formula.variable: t}):
217                 return True
218             return False
219
220 def is_model_of(self, formulas: AbstractSet[Formula]) -> bool:
221     """Checks if the current model is a model for the given formulas.
222
223     Returns:
224     ``True`` if each of the given formulas evaluates to true in the
225     current model for any assignment of elements from the universe of
226     the current model to the free occurrences of variables in that
227     formula, ``False`` otherwise.
228     """
229     for formula in formulas:
230         assert formula.constants().issubset(self.constant_meanings.keys())
231         for function, arity in formula.functions():
232             assert function in self.function_meanings and \
233                    self.function_aritys[function] == arity
234         for relation, arity in formula.relations():
235             assert relation in self.relation_meanings and \
236                    self.relation_aritys[relation] in {-1, arity}
237
238     # Task 7.9
239     free_vars = set()
240     for f in formulas:
241         free_vars.update(f.free_variables())
242     for assignment in product(self.universe, repeat=len(free_vars)):
243         params = dict(zip(free_vars, assignment))
244         for f in formulas:
245             if not self.evaluate_formula(f, params):
246                 return False
247     return True

```

## 4 code/predicates/syntax.py

```
1  # This file is part of the materials accompanying the book
2  # "Mathematical Logic through Python" by Gonczarowski and Nisan,
3  # Cambridge University Press. Book site: www.LogicThruPython.org
4  # (c) Yannai A. Gonczarowski and Noam Nisan, 2017-2020
5  # File name: predicates/syntax.py
6
7  """Syntactic handling of predicate-logic expressions."""
8
9  from __future__ import annotations
10 from typing import AbstractSet, Mapping, Optional, Sequence, Set, Tuple, Union
11
12 from logic_utils import fresh_variable_name_generator, frozen, \
13     memoized_parameterless_method
14
15 from propositions.syntax import Formula as PropositionalFormula, \
16     is_variable as is_propositional_variable
17 from functools import lru_cache
18
19
20 class ForbiddenVariableError(Exception):
21     """Raised by `Term.substitute` and `Formula.substitute` when a substituted
22     term contains a variable name that is forbidden in that context.
23
24     Attributes:
25         variable_name (`str`): the variable name that was forbidden in the
26             context in which a term containing it was to be substituted.
27     """
28     variable_name: str
29
30     def __init__(self, variable_name: str):
31         """Initializes a `ForbiddenVariableError` from the offending variable
32         name.
33
34         Parameters:
35             variable_name: variable name that is forbidden in the context in
36                 which a term containing it is to be substituted.
37         """
38         assert is_variable(variable_name)
39         self.variable_name = variable_name
40
41
42 @lru_cache(maxsize=100) # Cache the return value of is_constant
43 def is_constant(string: str) -> bool:
44     """Checks if the given string is a constant name.
45
46     Parameters:
47         string: string to check.
48
49     Returns:
50         ``True`` if the given string is a constant name, ``False`` otherwise.
51     """
52     return (((string[0] >= '0' and string[0] <= '9') or \
53             (string[0] >= 'a' and string[0] <= 'd')) and \
54            string.isalnum()) or string == '_'
55
56
57 @lru_cache(maxsize=100) # Cache the return value of is_variable
58 def is_variable(string: str) -> bool:
59     """Checks if the given string is a variable name.
```



```

60
61     Parameters:
62         string: string to check.
63
64     Returns:
65         ``True`` if the given string is a variable name, ``False`` otherwise.
66     """
67     return string[0] >= 'u' and string[0] <= 'z' and string.isalnum()
68
69
70 @lru_cache(maxsize=100) # Cache the return value of is_function
71 def is_function(string: str) -> bool:
72     """Checks if the given string is a function name.
73
74     Parameters:
75         string: string to check.
76
77     Returns:
78         ``True`` if the given string is a function name, ``False`` otherwise.
79     """
80     return string[0] >= 'f' and string[0] <= 't' and string.isalnum()
81
82
83 def find_relevant_part(string: str, func):
84     i = 1
85     while i < len(string) + 1:
86         if func(string[0:i]):
87             i += 1
88         else:
89             i -= 1
90             break
91     return i
92
93
94 @frozen
95 class Term:
96     """An immutable predicate-logic term in tree representation, composed from
97     variable names and constant names, and function names applied to them.
98
99     Attributes:
100         root (`str`): the constant name, variable name, or function name at the
101         root of the term tree.
102         arguments (~typing.Optional`\\[~typing.Tuple`\\[`Term`, ...]]): the
103         arguments to the root, if the root is a function name.
104     """
105     root: str
106     arguments: Optional[Tuple[Term, ...]]
107
108     def __init__(self, root: str, arguments: Optional[Sequence[Term]] = None):
109         """Initializes a `Term` from its root and root arguments.
110
111         Parameters:
112             root: the root for the formula tree.
113             arguments: the arguments to the root, if the root is a function
114             name.
115         """
116         if is_constant(root) or is_variable(root):
117             assert arguments is None
118             self.root = root
119         else:
120             assert is_function(root)
121             assert arguments is not None
122             self.root = root
123             self.arguments = tuple(arguments)
124             assert len(self.arguments) > 0
125
126     @memoized_parameterless_method
127     def __repr__(self) -> str:

```

```

128         """Computes the string representation of the current term.
129
130     Returns:
131         The standard string representation of the current term.
132     """
133     # Task 7.1
134     if not is_function(self.root):
135         return self.root
136     else:
137         return self.root + "(" + ",".join([str(x) for x in
138                                             self.arguments]) + ")"
139
140 def __eq__(self, other: object) -> bool:
141     """Compares the current term with the given one.
142
143     Parameters:
144         other: object to compare to.
145
146     Returns:
147         ``True`` if the given object is a `Term` object that equals the
148         current term, ``False`` otherwise.
149     """
150     return isinstance(other, Term) and str(self) == str(other)
151
152 def __ne__(self, other: object) -> bool:
153     """Compares the current term with the given one.
154
155     Parameters:
156         other: object to compare to.
157
158     Returns:
159         ``True`` if the given object is not a `Term` object or does not
160         equal the current term, ``False`` otherwise.
161     """
162     return not self == other
163
164 def __hash__(self) -> int:
165     return hash(str(self))
166
167 @staticmethod
168 def _parse_prefix(string: str) -> Tuple[Term, str]:
169     """Parses a prefix of the given string into a term.
170
171     Parameters:
172         string: string to parse, which has a prefix that is a valid
173         representation of a term.
174
175     Returns:
176         A pair of the parsed term and the unparsed suffix of the string. If
177         the given string has as a prefix a constant name (e.g., ``c12``)
178         or a variable name (e.g., ``x12``), then the parsed prefix will be
179         that entire name (and not just a part of it, such as ``x1``).
180     """
181     # Task 7.3.1
182     if is_variable(string[0]) or is_constant(string[0]):
183         i = find_relevant_part(string, lambda s: is_constant(s) or
184                                is_variable(s))
185
186         return Term(string[:i]), string[i:]
187     elif is_function(string[0]):
188         terms_lst = []
189         i = string.index('(')
190         t, rest = Term._parse_prefix(string[i + 1:])
191         terms_lst.append(t)
192         while rest[0] == ',':
193             t, rest = Term._parse_prefix(rest[1:])
194             terms_lst.append(t)
195         return Term(string[:i], terms_lst), rest[1:]

```

```

196
197 @staticmethod
198 def parse(string: str) -> Term:
199     """Parses the given valid string representation into a term.
200
201     Parameters:
202         string: string to parse.
203
204     Returns:
205         A term whose standard string representation is the given string.
206     """
207     # Task 7.3.2
208     prefix, suffix = Term._parse_prefix(string)
209     assert prefix is not None and len(suffix) == 0
210     return prefix
211
212 def __collect_vars(self, final_set, func):
213     if is_function(self.root):
214         if func(self.root):
215             final_set.add((self.root, len(self.arguments)))
216             for arg in self.arguments:
217                 arg.__collect_vars(final_set, func)
218         elif func(self.root):
219             final_set.add(self.root)
220         else:
221             return
222
223 @memoized_parameterless_method
224 def constants(self) -> Set[str]:
225     """Finds all constant names in the current term.
226
227     Returns:
228         A set of all constant names used in the current term.
229     """
230     # Task 7.5.1
231     final_set = set()
232     self.__collect_vars(final_set, is_constant)
233     return final_set
234
235 @memoized_parameterless_method
236 def variables(self) -> Set[str]:
237     """Finds all variable names in the current term.
238
239     Returns:
240         A set of all variable names used in the current term.
241     """
242     # Task 7.5.2
243     final_set = set()
244     self.__collect_vars(final_set, is_variable)
245     return final_set
246
247 @memoized_parameterless_method
248 def functions(self) -> Set[Tuple[str, int]]:
249     """Finds all function names in the current term, along with their
250     arities.
251
252     Returns:
253         A set of pairs of function name and arity (number of arguments) for
254         all function names used in the current term.
255     """
256     # Task 7.5.3
257     final_set = set()
258     self.__collect_vars(final_set, is_function)
259     return final_set
260
261 def substitute(self, substitution_map: Mapping[str, Term],
262               forbidden_variables: AbstractSet[str] = frozenset()) -> Term:
263     """Substitutes in the current term, each constant name `name` or

```

```

264     variable name `name` that is a key in `substitution_map` with the term
265     `substitution_map`\ ``[\ `name\ ``]\ ``.
266
267     Parameters:
268         substitution_map: mapping defining the substitutions to be
269         performed.
270         forbidden_variables: variables not allowed in substitution terms.
271
272     Returns:
273         The term resulting from performing all substitutions. Only
274         constant names and variable names originating in the current term
275         are substituted (i.e., those originating in one of the specified
276         substitutions are not subjected to additional substitutions).
277
278     Raises:
279         ForbiddenVariableError: If a term that is used in the requested
280         substitution contains a variable from `forbidden_variables`.
281
282     Examples:
283         >>> Term.parse('f(x,c)').substitute(
284         ...     {'c': Term.parse('plus(d,x)'), 'x': Term.parse('c')}, {'y'})
285         f(c,plus(d,x))
286
287         >>> Term.parse('f(x,c)').substitute(
288         ...     {'c': Term.parse('plus(d,y)'), 'y': Term.parse('c')}, {'y'})
289         Traceback (most recent call last):
290         ...
291         predicates.syntax.ForbiddenVariableError: y
292     """
293     for element_name in substitution_map:
294         assert is_constant(element_name) or is_variable(element_name)
295     for variable in forbidden_variables:
296         assert is_variable(variable)
297     # Task 9.1
298     for val in substitution_map.values():
299         intersection = val.variables().intersection(forbidden_variables)
300         if intersection:
301             raise ForbiddenVariableError(next(iter(intersection)))
302     return self.__substitute_helper(substitution_map)
303
304     def __substitute_helper(self, substitution_map: Mapping[str, Term]) -> Term:
305         if is_constant(self.root) or is_variable(self.root):
306             temp = substitution_map.get(self.root)
307             if temp is not None:
308                 return temp
309             return self
310         else:
311             return Term(self.root,
312                         [s.__substitute_helper(substitution_map) for
313                          s in self.arguments])
314
315     @lru_cache(maxsize=100) # Cache the return value of is_equality
316     def is_equality(string: str) -> bool:
317         """Checks if the given string is the equality relation.
318
319         Parameters:
320             string: string to check.
321
322         Returns:
323             ``True`` if the given string is the equality relation, ``False``
324             otherwise.
325         """
326         return string == '='
327
328     @lru_cache(maxsize=100) # Cache the return value of is_relation
329     def is_relation(string: str) -> bool:

```

```

332     """Checks if the given string is a relation name.
333
334     Parameters:
335         string: string to check.
336
337     Returns:
338         ``True`` if the given string is a relation name, ``False`` otherwise.
339     """
340     return string[0] >= 'F' and string[0] <= 'T' and string.isalnum()
341
342
343 @lru_cache(maxsize=100) # Cache the return value of is_unary
344 def is_unary(string: str) -> bool:
345     """Checks if the given string is a unary operator.
346
347     Parameters:
348         string: string to check.
349
350     Returns:
351         ``True`` if the given string is a unary operator, ``False`` otherwise.
352     """
353     return string == '~'
354
355
356 @lru_cache(maxsize=100) # Cache the return value of is_binary
357 def is_binary(string: str) -> bool:
358     """Checks if the given string is a binary operator.
359
360     Parameters:
361         string: string to check.
362
363     Returns:
364         ``True`` if the given string is a binary operator, ``False`` otherwise.
365     """
366     return string == '&' or string == '|' or string == '->'
367
368
369 @lru_cache(maxsize=100) # Cache the return value of is_quantifier
370 def is_quantifier(string: str) -> bool:
371     """Checks if the given string is a quantifier.
372
373     Parameters:
374         string: string to check.
375
376     Returns:
377         ``True`` if the given string is a quantifier, ``False`` otherwise.
378     """
379     return string == 'A' or string == 'E'
380
381
382 @frozen
383 class Formula:
384     """An immutable predicate-logic formula in tree representation, composed
385     from relation names applied to predicate-logic terms, and operators and
386     quantifications applied to them.
387
388     Attributes:
389         root (`str`): the relation name, equality relation, operator, or
390         quantifier at the root of the formula tree.
391         arguments (~typing.Optional`\\`~typing.Tuple`\\`[Term, ...]): the
392         arguments to the root, if the root is a relation name or the
393         equality relation.
394         first (~typing.Optional`\\`[Formula]): the first operand to the root,
395         if the root is a unary or binary operator.
396         second (~typing.Optional`\\`[Formula]): the second
397         operand to the root, if the root is a binary operator.
398         variable (~typing.Optional`\\`[str]): the variable name quantified by
399         the root, if the root is a quantification.

```

```

400         predicate (~typing.Optional`\\[Formula`]): the predicate quantified by
401             the root, if the root is a quantification.
402     """
403     root: str
404     arguments: Optional[Tuple[Term, ...]]
405     first: Optional[Formula]
406     second: Optional[Formula]
407     variable: Optional[str]
408     predicate: Optional[Formula]
409
410     def __init__(self, root: str,
411                 arguments_or_first_or_variable: Union[Sequence[Term],
412                                                         Formula, str],
413                 second_or_predicate: Optional[Formula] = None):
414         """Initializes a `Formula` from its root and root arguments, root
415             operands, or root quantified variable and predicate.
416
417         Parameters:
418             root: the root for the formula tree.
419             arguments_or_first_or_variable: the arguments to the root, if the
420                 root is a relation name or the equality relation; the first
421                 operand to the root, if the root is a unary or binary operator;
422                 the variable name quantified by the root, if the root is a
423                 quantification.
424             second_or_predicate: the second operand to the root, if the root is
425                 a binary operator; the predicate quantified by the root, if the
426                 root is a quantification.
427         """
428         if is_equality(root) or is_relation(root):
429             # Populate self.root and self.arguments
430             assert second_or_predicate is None
431             assert isinstance(arguments_or_first_or_variable, Sequence) and \
432                 not isinstance(arguments_or_first_or_variable, str)
433             self.root, self.arguments = \
434                 root, tuple(arguments_or_first_or_variable)
435             if is_equality(root):
436                 assert len(self.arguments) == 2
437         elif is_unary(root):
438             # Populate self.first
439             assert isinstance(arguments_or_first_or_variable, Formula) and \
440                 second_or_predicate is None
441             self.root, self.first = root, arguments_or_first_or_variable
442         elif is_binary(root):
443             # Populate self.first and self.second
444             assert isinstance(arguments_or_first_or_variable, Formula) and \
445                 second_or_predicate is not None
446             self.root, self.first, self.second = \
447                 root, arguments_or_first_or_variable, second_or_predicate
448         else:
449             assert is_quantifier(root)
450             # Populate self.variable and self.predicate
451             assert isinstance(arguments_or_first_or_variable, str) and \
452                 is_variable(arguments_or_first_or_variable) and \
453                 second_or_predicate is not None
454             self.root, self.variable, self.predicate = \
455                 root, arguments_or_first_or_variable, second_or_predicate
456
457     @memoized_parameterless_method
458     def __repr__(self) -> str:
459         """Computes the string representation of the current formula.
460
461         Returns:
462             The standard string representation of the current formula.
463         """
464         # Task 7.2
465         if is_equality(self.root):
466             return str(self.arguments[0]) + "=" + str(self.arguments[1])
467         elif is_relation(self.root):

```

```

468         return self.root + "(" + ",".join([str(x) for x in
469                                             self.arguments]) + ")"
470     elif is_unary(self.root):
471         return self.root + str(self.first)
472     elif is_binary(self.root):
473         return "(" + str(self.first) + self.root + str(self.second) + ")"
474     else:
475         # quantifier
476         return self.root + self.variable + "[" + str(self.predicate) + "]"
477
478 def __eq__(self, other: object) -> bool:
479     """Compares the current formula with the given one.
480
481     Parameters:
482         other: object to compare to.
483
484     Returns:
485         ``True`` if the given object is a `Formula` object that equals the
486         current formula, ``False`` otherwise.
487     """
488     return isinstance(other, Formula) and str(self) == str(other)
489
490 def __ne__(self, other: object) -> bool:
491     """Compares the current formula with the given one.
492
493     Parameters:
494         other: object to compare to.
495
496     Returns:
497         ``True`` if the given object is not a `Formula` object or does not
498         equal the current formula, ``False`` otherwise.
499     """
500     return not self == other
501
502 def __hash__(self) -> int:
503     return hash(str(self))
504
505 @staticmethod
506 def _parse_prefix(string: str) -> Tuple[Formula, str]:
507     """Parses a prefix of the given string into a formula.
508
509     Parameters:
510         string: string to parse, which has a prefix that is a valid
511         representation of a formula.
512
513     Returns:
514         A pair of the parsed formula and the unparsed suffix of the string.
515         If the given string has as a prefix a term followed by an equality
516         followed by a constant name (e.g., ``'c12'``) or by a variable name
517         (e.g., ``'x12'``), then the parsed prefix will include that entire
518         name (and not just a part of it, such as ``'x1'``).
519     """
520     # Task 7.4.1
521
522     if is_relation(string[0]):
523         terms = []
524         i = string.index('(')
525         if string[i + 1] != ')':
526             t, rest = Term._parse_prefix(string[i + 1:])
527             terms.append(t)
528             while rest[0] == ',':
529                 t, rest = Term._parse_prefix(rest[1:])
530                 terms.append(t)
531             rest = rest[1:]
532         else:
533             rest = string[i + 2:]
534         return Formula(string[:i], terms), rest
535     elif is_unary(string[0]):

```

```

536         f, rest = Formula._parse_prefix(string[1:])
537         return Formula('~', f), rest
538     elif string[0] == '(':
539         first, rest = Formula._parse_prefix(string[1:])
540         if is_binary(rest[0]):
541             i = 1
542         else:
543             i = 2
544         operator, rest = rest[:i], rest[i:]
545         second, rest2 = Formula._parse_prefix(rest)
546         return Formula(operator, first, second), rest2[1:]
547     elif is_quantifier(string[0]):
548         i = string.index('(')
549         var_name = string[1:i]
550         f, rest = Formula._parse_prefix(string[i + 1:])
551         return Formula(string[0], var_name, f), rest[1:]
552     else:
553         # i = string.index('=')
554         t1, rest = Term._parse_prefix(string)
555         t2, rest = Term._parse_prefix(rest[1:])
556         return Formula('=', [t1, t2]), rest
557
558 @staticmethod
559 def parse(string: str) -> Formula:
560     """Parses the given valid string representation into a formula.
561
562     Parameters:
563         string: string to parse.
564
565     Returns:
566         A formula whose standard string representation is the given string.
567     """
568     parsed, rest = Formula._parse_prefix(string)
569     assert rest is not None
570     return parsed
571
572 @memoized_parameterless_method
573 def constants(self) -> Set[str]:
574     """Finds all constant names in the current formula.
575
576     Returns:
577         A set of all constant names used in the current formula.
578     """
579     # Task 7.6.1
580     if is_equality(self.root) or is_relation(self.root):
581         s = set()
582         for term in self.arguments:
583             s.update(term.constants())
584         return s
585     elif is_unary(self.root):
586         return self.first.constants()
587     elif is_binary(self.root):
588         return self.first.constants().union(self.second.constants())
589     else:
590         # quantifier
591         return self.predicate.constants()
592
593 @memoized_parameterless_method
594 def variables(self) -> Set[str]:
595     """Finds all variable names in the current formula.
596
597     Returns:
598         A set of all variable names used in the current formula.
599     """
600     # Task 7.6.2
601     if is_equality(self.root) or is_relation(self.root):
602         s = set()
603         for term in self.arguments:

```



```

604         s.update(term.variables())
605     return s
606 elif is_unary(self.root):
607     return self.first.variables()
608 elif is_binary(self.root):
609     return self.first.variables().union(self.second.variables())
610 else:
611     # quantifier
612     return self.predicate.variables().union({self.variable})
613
614 @memoized_parameterless_method
615 def free_variables(self) -> Set[str]:
616     """Finds all variable names that are free in the current formula.
617
618     Returns:
619     A set of every variable name that is used in the current formula not
620     only within a scope of a quantification on that variable name.
621     """
622     # Task 7.6.3
623     if is_equality(self.root) or is_relation(self.root):
624         s = set()
625         for term in self.arguments:
626             s.update(term.variables())
627         return s
628     elif is_unary(self.root):
629         return self.first.free_variables()
630     elif is_binary(self.root):
631         return self.first.free_variables().union(
632             self.second.free_variables())
633     else:
634         # quantifier
635         x = self.predicate.free_variables()
636         x.discard(self.variable)
637         return x
638
639 @memoized_parameterless_method
640 def functions(self) -> Set[Tuple[str, int]]:
641     """Finds all function names in the current formula, along with their
642     arities.
643
644     Returns:
645     A set of pairs of function name and arity (number of arguments) for
646     all function names used in the current formula.
647     """
648     # Task 7.6.4
649     if is_equality(self.root) or is_relation(self.root):
650         s = set()
651         for term in self.arguments:
652             s.update(term.functions())
653         return s
654     elif is_unary(self.root):
655         return self.first.functions()
656     elif is_binary(self.root):
657         return self.first.functions().union(self.second.functions())
658     else:
659         # quantifier
660         return self.predicate.functions()
661
662 @memoized_parameterless_method
663 def relations(self) -> Set[Tuple[str, int]]:
664     """Finds all relation names in the current formula, along with their
665     arities.
666
667     Returns:
668     A set of pairs of relation name and arity (number of arguments) for
669     all relation names used in the current formula.
670     """
671     # Task 7.6.5

```

```

672     if is_equality(self.root):
673         return set()
674     elif is_relation(self.root):
675         return {(self.root, len(self.arguments))}
676     elif is_unary(self.root):
677         return self.first.relations()
678     elif is_binary(self.root):
679         return self.first.relations().union(self.second.relations())
680     else:
681         # quantifier
682         return self.predicate.relations()
683
684 def substitute(self, substitution_map: Mapping[str, Term],
685               forbidden_variables: AbstractSet[str] = frozenset()) -> \
686     Formula:
687     """Substitutes in the current formula, each constant name `name` or free
688     occurrence of variable name `name` that is a key in `substitution_map`
689     with the term `substitution_map` \ ``[`` `name` \ ``]`".
690
691     Parameters:
692         substitution_map: mapping defining the substitutions to be
693         performed.
694         forbidden_variables: variables not allowed in substitution terms.
695
696     Returns:
697         The formula resulting from performing all substitutions. Only
698         constant names and variable names originating in the current formula
699         are substituted (i.e., those originating in one of the specified
700         substitutions are not subjected to additional substitutions).
701
702     Raises:
703         ForbiddenVariableError: If a term that is used in the requested
704         substitution contains a variable from `forbidden_variables`
705         or a variable occurrence that becomes bound when that term is
706         substituted into the current formula.
707
708     Examples:
709         >>> Formula.parse('Ay[x=c]').substitute(
710         ...     {'c': Term.parse('plus(d,x)'), 'x': Term.parse('c')}, {'z'})
711         Ay[c=plus(d,x)]
712
713         >>> Formula.parse('Ay[x=c]').substitute(
714         ...     {'c': Term.parse('plus(d,z)'), {'z'})
715         Traceback (most recent call last):
716         ...
717         predicates.syntax.ForbiddenVariableError: z
718
719         >>> Formula.parse('Ay[x=c]').substitute(
720         ...     {'c': Term.parse('plus(d,y)'), {'z'})
721         Traceback (most recent call last):
722         ...
723         predicates.syntax.ForbiddenVariableError: y
724     """
725     for element_name in substitution_map:
726         assert is_constant(element_name) or is_variable(element_name)
727     for variable in forbidden_variables:
728         assert is_variable(variable)
729     # Task 9.2
730     return self.__substitute_formula_helper(forbidden_variables, substitution_map, self.free_variables())
731
732 def __substitute_formula_helper(self, forbidden_variables, substitution_map, free_vars):
733     if is_equality(self.root) or is_relation(self.root):
734         new_arguments = []
735         for arg in self.arguments:
736             if not arg.variables().issubset(free_vars):
737                 new_arguments.append(arg)
738             else:
739                 new_arguments.append(arg.substitute(substitution_map, forbidden_variables))

```

```

740         return Formula(self.root, new_arguments)
741     elif is_unary(self.root):
742         return Formula(self.root,
743                         self.first.substitute(substitution_map,
744                                              forbidden_variables))
745     elif is_binary(self.root):
746         return Formula(self.root,
747                         self.first.substitute(substitution_map,
748                                              forbidden_variables),
749                         self.second.substitute(substitution_map,
750                                              forbidden_variables))
751     else:
752         return Formula(self.root, self.variable,
753                         self.predicate.substitute(substitution_map,
754                                                    set(forbidden_variables).union({self.variable})))
755
756 def propositional_skeleton(self) -> Tuple[PropositionalFormula,
757                                           Mapping[str, Formula]]:
758     """Computes a propositional skeleton of the current formula.
759
760     Returns:
761     A pair. The first element of the pair is a propositional formula
762     obtained from the current formula by substituting every (outermost)
763     subformula that has a relation or quantifier at its root with an
764     atomic propositional formula, consistently such that multiple equal
765     such (outermost) subformulas are substituted with the same atomic
766     propositional formula. The atomic propositional formulas used for
767     substitution are obtained, from left to right, by calling
768     `next` \ ``(`` \ ``~logic_utils.fresh_variable_name_generator\ ``)``.
769     The second element of the pair is a mapping from each atomic
770     propositional formula to the subformula for which it was
771     substituted.
772
773     Examples:
774     >>> formula = Formula.parse('((Ax[x=7]&x=7)|(x=7->~Q(y)))')
775     >>> formula.propositional_skeleton()
776     (((z1&z2)|(z2->~z3)), {'z1': Ax[x=7], 'z2': x=7, 'z3': Q(y)})
777     >>> formula.propositional_skeleton()
778     (((z4&z5)|(z5->~z6)), {'z4': Ax[x=7], 'z5': x=7, 'z6': Q(y)})
779     """
780     # Task 9.8
781     var_map = dict()
782     return self.__skeleton_helper(var_map), {key: val for val, key in var_map.items()}
783
784 def __skeleton_helper(self, var_map):
785     if is_equality(self.root) or is_relation(self.root) or is_quantifier(self.root):
786         zi = var_map.get(self)
787         if not zi:
788             zi = next(fresh_variable_name_generator)
789             var_map.update({self: zi})
790         return PropositionalFormula(zi)
791     elif is_unary(self.root):
792         return PropositionalFormula(self.root, self.first.__skeleton_helper(var_map))
793     elif is_binary(self.root):
794         return PropositionalFormula(self.root, self.first.__skeleton_helper(var_map),
795                                     self.second.__skeleton_helper(var_map))
796
797 @staticmethod
798 def from_propositional_skeleton(skeleton: PropositionalFormula,
799                                substitution_map: Mapping[str, Formula]) -> \
800     Formula:
801     """Computes a predicate-logic formula from a propositional skeleton and
802     a substitution map.
803
804     Arguments:
805     skeleton: propositional skeleton for the formula to compute,
806     containing no constants or operators beyond ``~``, ``->``,
807     ``/``, and ``&``.

```

```

808         substitution_map: mapping from each atomic propositional subformula
809         of the given skeleton to a predicate-logic formula.
810
811     Returns:
812         A predicate-logic formula obtained from the given propositional
813         skeleton by substituting each atomic propositional subformula with
814         the formula mapped to it by the given map.
815
816     Examples:
817         >>> Formula.from_propositional_skeleton(
818             ...     PropositionalFormula.parse('((z1&z2)|(z2->~z3))'),
819             ...     {'z1': Formula.parse('Ax[x=7]'), 'z2': Formula.parse('x=7'),
820             ...     'z3': Formula.parse('Q(y)')}
821             ((Ax[x=7]&x=7)|(x=7->~Q(y)))
822         """
823     for operator in skeleton.operators():
824         assert is_unary(operator) or is_binary(operator)
825     for variable in skeleton.variables():
826         assert variable in substitution_map
827     # Task 9.10
828     if is_propositional_variable(skeleton.root):
829         return substitution_map[skeleton.root]
830     elif is_unary(skeleton.root):
831         return Formula(skeleton.root, Formula.from_propositional_skeleton(skeleton.first, substitution_map))
832     else:
833         return Formula(skeleton.root, Formula.from_propositional_skeleton(skeleton.first,
834                                                                           substitution_map),
835                         Formula.from_propositional_skeleton(skeleton.second, substitution_map))

```

## 5 code/propositions/syntax.py

```
1  # This file is part of the materials accompanying the book
2  # "Mathematical Logic through Python" by Gonczarowski and Nisan,
3  # Cambridge University Press. Book site: www.LogicThruPython.org
4  # (c) Yannai A. Gonczarowski and Noam Nisan, 2017-2020
5  # File name: propositions/syntax.py
6
7  """Syntactic handling of propositional formulas."""
8
9  from __future__ import annotations
10 from functools import lru_cache
11 from typing import Mapping, Optional, Set, Tuple, Union
12
13 from logic_utils import frozen, memoized_parameterless_method
14
15
16 @lru_cache(maxsize=100) # Cache the return value of is_variable
17 def is_variable(string: str) -> bool:
18     """Checks if the given string is an atomic proposition.
19
20     Parameters:
21         string: string to check.
22
23     Returns:
24         ``True`` if the given string is an atomic proposition, ``False``
25         otherwise.
26     """
27     return string[0] >= 'p' and string[0] <= 'z' and \
28         (len(string) == 1 or string[1:].isdigit())
29
30
31 @lru_cache(maxsize=100) # Cache the return value of is_constant
32 def is_constant(string: str) -> bool:
33     """Checks if the given string is a constant.
34
35     Parameters:
36         string: string to check.
37
38     Returns:
39         ``True`` if the given string is a constant, ``False`` otherwise.
40     """
41     return string == 'T' or string == 'F'
42
43
44 @lru_cache(maxsize=100) # Cache the return value of is_unary
45 def is_unary(string: str) -> bool:
46     """Checks if the given string is a unary operator.
47
48     Parameters:
49         string: string to check.
50
51     Returns:
52         ``True`` if the given string is a unary operator, ``False`` otherwise.
53     """
54     return string == '~'
55
56
57 @lru_cache(maxsize=100) # Cache the return value of is_binary
58 def is_binary(string: str) -> bool:
59     """Checks if the given string is a binary operator.
```

```

60
61     Parameters:
62         string: string to check.
63
64     Returns:
65         ``True`` if the given string is a binary operator, ``False`` otherwise.
66     """
67     # return string == '&' or string == '/' or string == '->'
68     # For Chapter 3:
69     return string in {'&', '|', '->', '+', '<->', '-&', '-|'}
70
71
72 @frozen
73 class Formula:
74     """An immutable propositional formula in tree representation, composed from
75     atomic propositions, and operators applied to them.
76
77     Attributes:
78         root (~str): the constant, atomic proposition, or operator at the root
79         of the formula tree.
80         first (~typing.Optional[Formula]): the first operand to the root,
81         if the root is a unary or binary operator.
82         second (~typing.Optional[Formula]): the second operand to the
83         root, if the root is a binary operator.
84     """
85     root: str
86     first: Optional[Formula]
87     second: Optional[Formula]
88
89     def __init__(self, root: str, first: Optional[Formula] = None,
90                 second: Optional[Formula] = None):
91         """Initializes a `Formula` from its root and root operands.
92
93         Parameters:
94             root: the root for the formula tree.
95             first: the first operand to the root, if the root is a unary or
96             binary operator.
97             second: the second operand to the root, if the root is a binary
98             operator.
99         """
100         if is_variable(root) or is_constant(root):
101             assert first is None and second is None
102             self.root = root
103         elif is_unary(root):
104             assert first is not None and second is None
105             self.root, self.first = root, first
106         else:
107             assert is_binary(root)
108             assert first is not None and second is not None
109             self.root, self.first, self.second = root, first, second
110
111     @memoized_parameterless_method
112     def __repr__(self) -> str:
113         """Computes the string representation of the current formula.
114
115         Returns:
116             The standard string representation of the current formula.
117         """
118         if is_variable(self.root) or is_constant(self.root):
119             return self.root
120         elif is_unary(self.root):
121             return self.root + str(self.first)
122         else:
123             return "(" + str(self.first) + self.root + str(self.second) + ")"
124
125     def __eq__(self, other: object) -> bool:
126         """Compares the current formula with the given one.
127

```

```

128     Parameters:
129         other: object to compare to.
130
131     Returns:
132         ``True`` if the given object is a `Formula` object that equals the
133         current formula, ``False`` otherwise.
134     """
135     return isinstance(other, Formula) and str(self) == str(other)
136
137 def __ne__(self, other: object) -> bool:
138     """Compares the current formula with the given one.
139
140     Parameters:
141         other: object to compare to.
142
143     Returns:
144         ``True`` if the given object is not a `Formula` object or does not
145         equal the current formula, ``False`` otherwise.
146     """
147     return not self == other
148
149 def __hash__(self) -> int:
150     return hash(str(self))
151
152 @memoized_parameterless_method
153 def variables(self) -> Set[str]:
154     """Finds all atomic propositions (variables) in the current formula.
155
156     Returns:
157         A set of all atomic propositions used in the current formula.
158     """
159     if is_variable(self.root):
160         return {self.root}
161     elif is_constant(self.root):
162         return set()
163     elif is_unary(self.root):
164         return self.first.variables()
165     else:
166         return self.first.variables().union(self.second.variables())
167
168 @memoized_parameterless_method
169 def operators(self) -> Set[str]:
170     """Finds all operators in the current formula.
171
172     Returns:
173         A set of all operators (including ``'T'`` and ``'F'``) used in the
174         current formula.
175     """
176     if is_variable(self.root):
177         return set()
178     elif is_constant(self.root):
179         return {self.root}
180     elif is_unary(self.root):
181         return {self.root}.union(self.first.operators())
182     else:
183         return {self.root}.union(self.first.operators(), self.second.operators())
184
185 @staticmethod
186 def _parse_prefix(string: str) -> Tuple[Union[Formula, None], str]:
187     """Parses a prefix of the given string into a formula.
188
189     Parameters:
190         string: string to parse.
191
192     Returns:
193         A pair of the parsed formula and the unparsed suffix of the string.
194         If the given string has as a prefix a variable name (e.g.,
195         ``'x12'``) or a unary operator follows by a variable name, then the

```

```

196         parsed prefix will include that entire variable name (and not just a
197         part of it, such as ``x1``). If no prefix of the given string is a
198         valid standard string representation of a formula then returned pair
199         should be of ``None`` and an error message, where the error message
200         is a string with some human-readable content.
201     """
202     if len(string) == 0:
203         return None, "Formula is empty."
204     elif is_constant(string[0]):
205         return Formula(string[0]), string[1:]
206     elif is_variable(string[0]):
207         i = 1
208         while i < len(string) + 1:
209             if is_variable(string[0:i]):
210                 i += 1
211             else:
212                 i -= 1
213             break
214         return Formula(string[0:i]), string[i:]
215     elif string[0] == "(":
216         first, rest = Formula._parse_prefix(string[1:])
217         operator = ""
218         rest2 = rest
219         i = 0
220         while (not is_binary(operator)) and i < len(rest2):
221             operator = rest2[0:i]
222             rest2 = rest2[i:]
223             i = i + 1
224
225         if not is_binary(operator):
226             return None, "Binary operators must be one of: &, |, ->"
227
228         second, rest2 = Formula._parse_prefix(rest2)
229         if first is None or second is None or len(rest2) == 0 or rest2[0] != ")":
230             return None, "The use of binary operator is: '(<valid formula1>*<valid formula2>)', where * is" \
231                 " the binary operator."
232
233         return Formula(operator, first, second), rest2[1:]
234     elif is_unary(string[0]):
235         f, rest = Formula._parse_prefix(string[1:])
236         if f is None:
237             return None, "The use of not operator is: '~<valid formula>'"
238         return Formula(string[0], f), rest
239     else:
240         return None, "Expected valid formula."
241
242     @staticmethod
243     def is_formula(string: str) -> bool:
244         """Checks if the given string is a valid representation of a formula.
245
246         Parameters:
247             string: string to check.
248
249         Returns:
250             ``True`` if the given string is a valid standard string
251             representation of a formula, ``False`` otherwise.
252         """
253         prefix, suffix = Formula._parse_prefix(string)
254         return prefix is not None and len(suffix) == 0
255
256     @staticmethod
257     def parse(string: str) -> Formula:
258         """Parses the given valid string representation into a formula.
259
260         Parameters:
261             string: string to parse.
262
263         Returns:

```



```

264         A formula whose standard string representation is the given string.
265         """
266         prefix, suffix = Formula._parse_prefix(string)
267         assert prefix is not None and len(suffix) == 0
268         return prefix
269
270     # Optional tasks for Chapter 1
271
272     def polish(self) -> str:
273         """Computes the polish notation representation of the current formula.
274
275         Returns:
276         The polish notation representation of the current formula.
277         """
278         if is_variable(self.root) or is_constant(self.root):
279             return self.root
280         elif is_unary(self.root):
281             return self.root + self.first.polish()
282         else:
283             return self.root + self.first.polish() + self.second.polish()
284
285     @staticmethod
286     def _parse_polish_prefix(string: str) -> Tuple[Union[Formula, None], str]:
287         if is_constant(string[0]):
288             return Formula(string[0]), string[1:]
289         elif is_variable(string[0]):
290             i = 1
291             while i < len(string) + 1:
292                 if is_variable(string[0:i]):
293                     i += 1
294                 else:
295                     i -= 1
296             break
297             return Formula(string[0:i]), string[i:]
298         elif is_unary(string[0]):
299             f, rest = Formula._parse_polish_prefix(string[1:])
300             if f is None:
301                 return None, "The use of not operator is: '~<valid formula>'"
302             return Formula(string[0], f), rest
303         else:
304             if is_binary(string[0]):
305                 operator = string[0]
306                 string = string[1:]
307             elif is_binary(string[0:2]):
308                 operator = string[0:2]
309                 string = string[2:]
310             else:
311                 return None, "Expected valid formula."
312             first, rest1 = Formula._parse_polish_prefix(string)
313             second, rest2 = Formula._parse_polish_prefix(rest1)
314             if first is None or second is None:
315                 return None, "The use of binary operator is: *<f1><f2>"
316             return Formula(operator, first, second), rest2
317
318     @staticmethod
319     def parse_polish(string: str) -> Formula:
320         """Parses the given polish notation representation into a formula.
321
322         Parameters:
323         string: string to parse.
324
325         Returns:
326         A formula whose polish notation representation is the given string.
327         """
328         return Formula._parse_polish_prefix(string)[0]
329
330     def substitute_variables(self, substitution_map: Mapping[str, Formula]) -> \
331         Formula:

```

```

332     """Substitutes in the current formula, each variable `v` that is a key
333     in `substitution_map` with the formula `substitution_map[v]`.
334
335     Parameters:
336         substitution_map: mapping defining the substitutions to be
337         performed.
338
339     Returns:
340         The formula resulting from performing all substitutions. Only
341         variables originating in the current formula are substituted (i.e.,
342         variables originating in one of the specified substitutions are not
343         subjected to additional substitutions).
344
345     Examples:
346         >>> Formula.parse('((p->p)|r)').substitute_variables(
347             ...     {'p': Formula.parse('q&r'), 'r': Formula.parse('p')})
348         (((q&r)->(q&r))|p)
349     """
350     for variable in substitution_map:
351         assert is_variable(variable)
352     if is_variable(self.root) or is_constant(self.root):
353         return substitution_map.get(self.root, self)
354     if is_unary(self.root):
355         return Formula(self.root, self.first.substitute_variables(substitution_map))
356     if is_binary(self.root):
357         return Formula(self.root, self.first.substitute_variables(substitution_map),
358             self.second.substitute_variables(substitution_map))
359
360 def substitute_operators(self, substitution_map: Mapping[str, Formula]) -> \
361     Formula:
362     """Substitutes in the current formula, each constant or operator `op`
363     that is a key in `substitution_map` with the formula
364     `substitution_map[op]` applied to its (zero or one or two) operands,
365     where the first operand is used for every occurrence of ``'p'`` in the
366     formula and the second for every occurrence of ``'q'``.
367
368     Parameters:
369         substitution_map: mapping defining the substitutions to be
370         performed.
371
372     Returns:
373         The formula resulting from performing all substitutions. Only
374         operators originating in the current formula are substituted (i.e.,
375         operators originating in one of the specified substitutions are not
376         subjected to additional substitutions).
377
378     Examples:
379         >>> Formula.parse('((x&y)&-z)').substitute_operators(
380             ...     {'&': Formula.parse('~(p|~q)')})
381         ~(~(x|~y)|~z)
382     """
383     for operator in substitution_map:
384         assert is_binary(operator) or is_unary(operator) or \
385             is_constant(operator)
386         assert substitution_map[operator].variables().issubset({'p', 'q'})
387
388     p = Formula("p")
389     q = Formula("q")
390     if is_variable(self.root):
391         return self
392     if is_constant(self.root):
393         return substitution_map.get(self.root, self)
394     if is_unary(self.root):
395         _map = {'p': self.first.substitute_operators(substitution_map)}
396         return substitution_map.get(self.root, Formula(self.root, p)).substitute_variables(_map)
397     if is_binary(self.root):
398         _map = {'p': self.first.substitute_operators(substitution_map),
399             'q': self.second.substitute_operators(substitution_map)}

```

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