

Contents

1	Basic Test Results	2
2	README	3
3	code/predicates/semantics.py	4
4	code/predicates/syntax.py	8
5	code/propositions/syntax.py	9

1 Basic Test Results

```
1  unzipping /tmp/bodek.woFmwe/logic/ex7/alonzo/presubmission/submission
2  Archive:  /tmp/bodek.woFmwe/logic/ex7/alonzo/presubmission/submission
3      creating: code/predicates/
4      inflating: code/predicates/semantics.py
5      inflating: code/predicates/syntax.py
6      creating: code/propositions/
7      inflating: code/propositions/syntax.py
8  extracting: README
9
10 searching for README:
11     README found!
12 usernames:
13 ['ori.frenkel', 'alonzo']
14
15 required files:
16 copying code/propositions/syntax.py
17 copying code/predicates/syntax.py
18 copying code/predicates/semantics.py
19
20 optional files:
21
22 test_task1 Passed
23 test_task2 Passed
24 test_task3 Passed
25 test_task4 Passed
26 test_task5 Passed
27 test_task6 Passed
28 test_task7 Passed
29 test_task8 Passed
30 test_task9 Passed
```

2 README

```
1  ori.frenkel
2  alonzo
3  ***
```

3 code/predicates/semantics.py

```
1  # (c) This file is part of the course
2  # Mathematical Logic through Programming
3  # by Gonczarowski and Nisan.
4  # File name: predicates/semantics.py
5
6  """Semantic analysis of first-order logic constructs."""
7
8  from typing import AbstractSet, FrozenSet, Generic, Mapping, Tuple, TypeVar
9
10 from logic_utils import frozen, frozendict
11
12 from predicates.syntax import *
13
14 import itertools
15
16 AND = '&'; OR = '|'; IMPLY = '->'
17 ALL = 'A'; EXISTS = 'E'
18 #: A generic type for a universe element in a model.
19 T = TypeVar('T')
20
21 @frozen
22 class Model(Generic[T]):
23     """An immutable model for first-order logic constructs.
24
25     Attributes:
26         universe (~typing.FrozenSet\\[T]): the set of elements to which
27             terms can be evaluated and over which quantifications are defined.
28         constant_meanings (~typing.Mapping\\[str, T]): mapping from each
29             constant name to the universe element to which it evaluates.
30         relation_aritys (~typing.Mapping\\[str, int]): mapping from
31             each relation name to the arity of the relation, or to -1 if the
32             relation is the empty relation.
33         relation_meanings (~typing.Mapping\\[str, ~typing.AbstractSet\\[~typing.Tuple\\[T, ...]]]):
34             mapping from each n-ary relation name to argument n-tuples (of
35             universe elements) for which the relation is true.
36         function_aritys (~typing.Mapping\\[str, int]): mapping from
37             each function name to the arity of the function.
38         function_meanings (~typing.Mapping\\[str, ~typing.Mapping\\[~typing.Tuple\\[T, ...], T]]):
39             mapping from each n-ary function name to the mapping from each
40             argument n-tuple (of universe elements) to the universe element that
41             the function outputs given these arguments.
42
43     """
44     universe: FrozenSet[T]
45     constant_meanings: Mapping[str, T]
46     relation_aritys: Mapping[str, int]
47     relation_meanings: Mapping[str, AbstractSet[Tuple[T, ...]]]
48     function_aritys: Mapping[str, int]
49     function_meanings: Mapping[str, Mapping[Tuple[T, ...], T]]
50
51     def __init__(self, universe: AbstractSet[T],
52                  constant_meanings: Mapping[str, T],
53                  relation_meanings: Mapping[str, AbstractSet[Tuple[T, ...]]],
54                  function_meanings: Mapping[str, Mapping[Tuple[T, ...], T]] =
55                      frozendict()) -> None:
56         """Initializes a Model from its universe and constant, relation, and
57             function meanings.
58
59         Parameters:
60             universe: the set of elements to which terms are to be evaluated
```

```

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

```

```

128
129 def evaluate_term(self, term: Term,
130                  assignment: Mapping[str, T] = frozendict()) -> T:
131     """Calculates the value of the given term in the current model, for the
132     given assignment of values to variables names.
133
134     Parameters:
135         term: term to calculate the value of, for the constants and
136         functions of which the current model has meanings.
137         assignment: mapping from each variable name in the given term to a
138         universe element to which it is to be evaluated.
139
140     Returns:
141         The value (in the universe of the current model) of the given
142         term in the current model, for the given assignment of values to
143         variable names.
144     """
145     assert term.constants().issubset(self.constant_meanings.keys())
146     assert term.variables().issubset(assignment.keys())
147     for function, arity in term.functions():
148         assert function in self.function_meanings and \
149             self.function_arities[function] == arity
150     # Task 7.7
151     if is_constant(term.root):
152         return self.constant_meanings[term.root]
153
154     elif is_variable(term.root):
155         return assignment[term.root]
156
157     elif is_function(term.root):
158         arguments_evalutaions = [self.evaluate_term(arg, assignment)
159                                 for arg in term.arguments]
160         return self.function_meanings[term.root][tuple(arguments_evalutaions)]
161
162 def evaluate_formula(self, formula: Formula,
163                    assignment: Mapping[str, T] = frozendict()) -> bool:
164     """Calculates the truth value of the given formula in the current model,
165     for the given assignment of values to free occurrences of variables
166     names.
167
168     Parameters:
169         formula: formula to calculate the truth value of, for the constants,
170         functions, and relations of which the current model has
171         meanings.
172         assignment: mapping from each variable name that has a free
173         occurrence in the given formula to a universe element to which
174         it is to be evaluated.
175
176     Returns:
177         The truth value of the given formula in the current model, for the
178         given assignment of values to free occurrences of variable names.
179     """
180     assert formula.constants().issubset(self.constant_meanings.keys())
181     assert formula.free_variables().issubset(assignment.keys())
182     for function,arity in formula.functions():
183         assert function in self.function_meanings and \
184             self.function_arities[function] == arity
185     for relation,arity in formula.relations():
186         assert relation in self.relation_meanings and \
187             self.relation_arities[relation] in {-1, arity}
188     # Task 7.8
189     if is_equality(formula.root):
190         return self.evaluate_term(formula.arguments[0], assignment) == \
191             self.evaluate_term(formula.arguments[1], assignment)
192
193     elif is_relation(formula.root):
194         terms_evalutaions = [self.evaluate_term(arg, assignment)
195                             for arg in formula.arguments]

```

```

196         return tuple(terms_evalutaions) in self.relation_meanings[formula.root]
197
198
199     elif is_unary(formula.root):
200         return not self.evaluate_formula(formula.first, assignment)
201
202     elif is_binary(formula.root):
203         f1 = self.evaluate_formula(formula.first, assignment)
204         f2 = self.evaluate_formula(formula.second, assignment)
205
206         if formula.root == AND:
207             return f1 and f2
208         elif formula.root == OR:
209             return f1 or f2
210         elif formula.root == IMPLY:
211             return (f1 and f2) or not f1
212
213     elif is_quantifier(formula.root):
214         if formula.root == ALL:
215             for element in self.universe:
216                 new_assignment = dict(assignment)
217                 new_assignment[formula.variable] = element
218                 # There exists one model where formula evaluates to False
219                 if not self.evaluate_formula(formula.predicate, new_assignment):
220                     return False
221                 # for all assignments of the variable of quantification,
222                 # formula evaluates to True
223             return True
224         elif formula.root == EXISTS:
225             for element in self.universe:
226                 new_assignment = dict(assignment)
227                 new_assignment[formula.variable] = element
228                 # There exists one model where formula evaluates to True
229                 if self.evaluate_formula(formula.predicate, new_assignment):
230                     return True
231                 # for all assignments of the variable of quantification,
232                 # formula evaluates to False
233             return False
234
235     def is_model_of(self, formulas: AbstractSet[Formula]) -> bool:
236         """Checks if the current model is a model for the given formulas.
237
238         Returns:
239             ``True`` if each of the given formulas evaluates to true in the
240             current model for any assignment of elements from the universe of
241             the current model to the free occurrences of variables in that
242             formula, ``False`` otherwise.
243         """
244         for formula in formulas:
245             assert formula.constants().issubset(self.constant_meanings.keys())
246             for function, arity in formula.functions():
247                 assert function in self.function_meanings and \
248                     self.function_arities[function] == arity
249             for relation, arity in formula.relations():
250                 assert relation in self.relation_meanings and \
251                     self.relation_arities[relation] in {-1, arity}
252
253         # Task 7.9
254         for formula in formulas:
255             variables = list(formula.free_variables())
256             for assignment in itertools.product(self.universe, repeat=len(variables)):
257                 new_assignment = {variables[i]: assignment[i] for i in range(len(variables))}
258                 if not self.evaluate_formula(formula, new_assignment):
259                     return False
260
261         # all formulas evaluate to true
262         return True

```

4 `code/predicates/syntax.py`

This file can't be converted to PDF because it has unknown mime type or charset. This is usually due to illegal characters.

- mime type: text/x-python
- charset: unknown-8bit

5 code/propositions/syntax.py

```
1  # (c) This file is part of the course
2  # Mathematical Logic through Programming
3  # by Gonczarowski and Nisan.
4  # File name: propositions/syntax.py
5
6  """Syntactic handling of propositional formulae."""
7
8  from __future__ import annotations
9  from typing import Mapping, Optional, Set, Tuple, Union
10
11  from logic_utils import frozen
12
13  EMPTY_STRING = ''
14  OPEN_BRACKET = '('
15  LEFT_BRACKET = 1
16  CLOSE_BRACKET = ')'
17  RIGHT_BRACKET = -1
18  UNEXPECTED_SYMBOL = 'Unexpected Symbol'
19
20
21  def is_variable(s: str) -> bool:
22      """Checks if the given string is an atomic proposition.
23
24      Parameters:
25      s: string to check.
26
27      Returns:
28      ``True`` if the given string is an atomic proposition, ``False``
29      otherwise.
30      """
31      return s[0] >= 'p' and s[0] <= 'z' and (len(s) == 1 or s[1:].isdigit())
32
33
34  def is_constant(s: str) -> bool:
35      """Checks if the given string is a constant.
36
37      Parameters:
38      s: string to check.
39
40      Returns:
41      ``True`` if the given string is a constant, ``False`` otherwise.
42      """
43      return s == 'T' or s == 'F'
44
45
46  def is_unary(s: str) -> bool:
47      """Checks if the given string is a unary operator.
48
49      Parameters:
50      s: string to check.
51
52      Returns:
53      ``True`` if the given string is a unary operator, ``False`` otherwise.
54      """
55      return s == '~'
56
57
58  def is_binary(s: str) -> bool:
59      """Checks if the given string is a binary operator.
```

```

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

```

```

128         does not equal the current formula, ``False`` otherwise.
129         """
130         return not self == other
131
132     def __hash__(self) -> int:
133         return hash(str(self))
134
135     def __repr__(self) -> str:
136         """Computes the string representation of the current formula.
137
138         Returns:
139             The standard string representation of the current formula.
140         """
141         if is_variable(self.root) or is_constant(self.root):
142             return self.root
143         if is_unary(self.root):
144             return self.root + repr(self.first)
145         if is_binary(self.root):
146             return '(' + repr(self.first) + self.root + repr(self.second) + ')'
147
148     def variables(self) -> Set[str]:
149         """Finds all atomic propositions (variables) in the current formula.
150
151         Returns:
152             A set of all atomic propositions used in the current formula.
153         """
154         if is_constant(self.root):
155             return set()
156         if is_variable(self.root):
157             return {self.root}
158         if is_unary(self.root):
159             return self.first.variables()
160         if is_binary(self.root):
161             return self.first.variables().union(self.second.variables())
162
163     def operators(self) -> Set[str]:
164         """Finds all operators in the current formula.
165
166         Returns:
167             A set of all operators (including ``'T'`` and ``'F'``) used in the
168             current formula.
169         """
170         if is_variable(self.root):
171             return set()
172         if is_constant(self.root):
173             return {self.root}
174         if is_unary(self.root):
175             return {self.root}.union(self.first.operators())
176         if is_binary(self.root):
177             operators = self.first.operators().union(self.second.operators())
178             return operators.union({self.root})
179
180     @staticmethod
181     def parse_prefix(s: str) -> Tuple[Union[Formula, None], str]:
182         """Parses a prefix of the given string into a formula.
183
184         Parameters:
185             s: string to parse.
186
187         Returns:
188             A pair of the parsed formula and the unparsed suffix of the string.
189             If the first token of the string is a variable name (e.g.,
190             ``'x12'``), then the parsed prefix will be that entire variable name
191             (and not just a part of it, such as ``'x1'``). If no prefix of the
192             given string is a valid standard string representation of a formula
193             then returned pair should be of ``None`` and an error message, where
194             the error message is a string with some human-readable content.
195         """

```

```

196     # check if given string is empty
197     if s == EMPTY_STRING:
198         return None, UNEXPECTED_SYMBOL
199
200     # Check for constant prefix
201     if is_constant(s[0]):
202         return Formula(s[0]), s[1:]
203
204     # Check for variable prefix
205     if is_variable(s[0]):
206         index = 1
207         while index < len(s) and is_variable(s[:index+1]):
208             index += 1
209         return Formula(s[:index]), s[index:]
210
211     # Check for unary prefix
212     if is_unary(s[0]):
213         parsed = Formula.parse_prefix(s[1:])
214         if parsed[1] != UNEXPECTED_SYMBOL:
215             return Formula(s[0], parsed[0]), parsed[1]
216
217     # Check for binary prefix
218     if s[0] == OPEN_BRACKET:
219         index = 1
220         # look for the binary operator and the two formulas in its sides
221         root_index_start = None
222         root_index_end = None
223         sum_brackets = LEFT_BRACKET
224         while sum_brackets > 0 and index < len(s):
225             # open bracket
226             if s[index] == OPEN_BRACKET:
227                 sum_brackets += LEFT_BRACKET
228             # close bracket
229             if s[index] == CLOSE_BRACKET:
230                 sum_brackets += RIGHT_BRACKET
231             # binary operation, length 1
232             if is_binary(s[index]) and sum_brackets == 1:
233                 if root_index_start is not None:
234                     return None, UNEXPECTED_SYMBOL
235                 root_index_start = index
236                 root_index_end = index + 1
237             # binary operation, length 2
238             if (index + 1) < len(s):
239                 if is_binary(s[index:index+2]) and sum_brackets == 1:
240                     if root_index_start is not None:
241                         return None, UNEXPECTED_SYMBOL
242                     root_index_start = index
243                     root_index_end = index + 2
244                     index += 2
245                     continue
246             # binary operation, length 3
247             if (index + 2) < len(s):
248                 if is_binary(s[index:index + 3]) and sum_brackets == 1:
249                     if root_index_start is not None:
250                         return None, UNEXPECTED_SYMBOL
251                     root_index_start = index
252                     root_index_end = index + 3
253                     index += 3
254                     continue
255             # else, one char was read, a bracket or a binary operation
256             index += 1
257
258     # Check if a binary operation was found
259     if root_index_start is None or sum_brackets > 0:
260         return None, UNEXPECTED_SYMBOL
261     operation = s[root_index_start:root_index_end]
262     parse_first = Formula.parse_prefix(s[1:root_index_start])
263     # Check if the formula on the left side of the operation is legal

```

```

264         if parse_first[1] != EMPTY_STRING:
265             return None, UNEXPECTED_SYMBOL
266         # Check if the formula on the right side of the operation is legal
267         parse_second = Formula.parse_prefix(s[root_index_end:index-1])
268         if parse_second[1] != EMPTY_STRING:
269             return None, UNEXPECTED_SYMBOL
270         returned_formula = \
271             Formula(operation, parse_first[0], parse_second[0])
272         formula_end = s[index:]
273         return returned_formula, formula_end
274
275     # else, the prefix doesn't fit any Formula patterns
276     return None, UNEXPECTED_SYMBOL
277
278
279 @staticmethod
280 def is_formula(s: str) -> bool:
281     """Checks if the given string is a valid representation of a formula.
282
283     Parameters:
284         s: string to check.
285
286     Returns:
287         ``True`` if the given string is a valid standard string
288         representation of a formula, ``False`` otherwise.
289     """
290     if Formula.parse_prefix(s)[1] == EMPTY_STRING:
291         return True
292
293 @staticmethod
294 def parse(s: str) -> Formula:
295     """Parses the given valid string representation into a formula.
296
297     Parameters:
298         s: string to parse.
299
300     Returns:
301         A formula whose standard string representation is the given string.
302     """
303     assert Formula.is_formula(s)
304     return Formula.parse_prefix(s)[0]
305
306 # Optional tasks for Chapter 1
307
308 def polish(self) -> str:
309     """Computes the polish notation representation of the current formula.
310
311     Returns:
312         The polish notation representation of the current formula.
313     """
314     # Optional Task 1.7
315
316 @staticmethod
317 def parse_polish(s: str) -> Formula:
318     """Parses the given polish notation representation into a formula.
319
320     Parameters:
321         s: string to parse.
322
323     Returns:
324         A formula whose polish notation representation is the given string.
325     """
326     # Optional Task 1.8
327
328 # Tasks for Chapter 3
329
330 def substitute_variables(
331     self, substitution_map: Mapping[str, Formula]) -> 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: the mapping defining the substitutions to be
337         performed.
338
339     Returns:
340         The resulting formula.
341
342     Examples:
343         >>> Formula.parse('((p->p)/z)').substitute_variables(
344         ...     {'p': Formula.parse('(q&r)')})
345         (((q&r)->(q&r))/z)
346     """
347     for variable in substitution_map:
348         assert is_variable(variable)
349     # root is a constant
350     if is_constant(self.root):
351         return self
352     # root is a variable
353     if is_variable(self.root):
354         if self.root in substitution_map:
355             return substitution_map[self.root]
356         # else, root is not a variable in the mapping
357         return self
358     # root is an unary operation
359     if is_unary(self.root):
360         first = Formula.substitute_variables(self.first, substitution_map)
361         return Formula(root=self.root, first=first)
362     # root is an binary operation
363     if is_binary(self.root):
364         f1 = Formula.substitute_variables(self.first, substitution_map)
365         f2 = Formula.substitute_variables(self.second, substitution_map)
366         return Formula(root=self.root, first=f1, second=f2)
367     # Task 3.3
368
369 def substitute_operators(
370     self, substitution_map: Mapping[str, Formula]) -> Formula:
371     """Substitutes in the current formula, each constant or operator `op`
372     that is a key in `substitution_map` with the formula
373     `substitution_map[op]` applied to its (zero or one or two) operands,
374     where the first operand is used for every occurrence of ``'p'`` in the
375     formula and the second for every occurrence of ``'q'``.
376
377     Parameters:
378         substitution_map: the mapping defining the substitutions to be
379         performed.
380
381     Returns:
382         The resulting formula.
383
384     Examples:
385         >>> Formula.parse('((x&y)&~z)').substitute_operators(
386         ...     {'&': Formula.parse('~(p/~q)')})
387         ~((~(x/~y)|~z)
388     """
389     for operator in substitution_map:
390         assert is_binary(operator) or is_unary(operator) or \
391             is_constant(operator)
392         assert substitution_map[operator].variables().issubset({'p', 'q'})
393
394     # root is a constant
395     if is_constant(self.root):
396         if self.root in substitution_map:
397             return substitution_map[self.root]
398         # else
399         return self

```

```

400
401     # root is a variable
402     if is_variable(self.root):
403         return self
404
405     # root is an unary operation
406     if is_unary(self.root):
407         f1 = Formula.substitute_operators(self.first, substitution_map)
408         if self.root in substitution_map:
409             f = substitution_map[self.root]
410             return Formula.substitute_variables(f, {'p': f1})
411         # else, operator not in substitution_map
412         return Formula(root=self.root, first=f1)
413
414     # root is a binary operation
415     if is_binary(self.root):
416         f1 = Formula.substitute_operators(self.first, substitution_map)
417         f2 = Formula.substitute_operators(self.second,
418                                         substitution_map)
419         if self.root in substitution_map:
420             f = substitution_map[self.root]
421             return Formula.substitute_variables(f, {'p': f1, 'q': f2})
422         # else, operator not in substitution_map
423         return Formula(root=self.root, first=f1, second=f2)
424     # Task 3.4

```