Contents

1 Basic Test Results

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

searching for README:
    README found!
usernames:
['ori.frenkel', 'alonzo']

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

optional files:

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

```
ori.frenkel
alonzo
***
```

3 code/predicates/semantics.py

```python
# (c) This file is part of the course
# Mathematical Logic through Programming
# by Gonczarowski and Nisan.
# File name: predicates/semantics.py

"""Semantic analysis of first-order logic constructs."""

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

from logic_utils import frozen, frozendict

from predicates.syntax import *

import itertools

AND = '&'; OR= '|' ; IMPLY = '->'
ALL = 'A'; EXISTS = 'E'
#: A generic type for a universe element in a model.
T = TypeVar('T')

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

    Attributes:
        universe (`~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()) -> None:
        """Initializes a `Model` from its universe and constant, relation, and
        function meanings.

        Parameters:
            universe: the set of elements to which terms are to be evaluated
```

```python
                and over which quantifications are to be defined.
            constant_meanings: mapping from each constant name to a universe
                element to which it is to be evaluated.
            relation_meanings: mapping from each relation name that is to
                be the name of an n-ary relation, to the argument n-tuples (of
                universe elements) for which the relation is to be true.
            function_meanings: mapping from each function name that is to
                be the name of an n-ary function, to a mapping from each
                argument n-tuple (of universe elements) to a universe element
                that the function is to output given these arguments.
        """
        self.universe = frozenset(universe)

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

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

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

    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 '')
```

```python

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

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

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

        elif is_variable(term.root):
            return assignment[term.root]

        elif is_function(term.root):
            arguments_evalutaions = [self.evaluate_term(arg, assignment)
            for arg in term.arguments]
            return self.function_meanings[term.root][tuple(arguments_evalutaions)]

    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):
            terms_evalutaions = [self.evaluate_term(arg, assignment)
                                 for arg in formula.arguments]
```

```python

            return tuple(terms_evalutaions) in self.relation_meanings[formula.root]

        elif is_unary(formula.root):
            return not self.evaluate_formula(formula.first, assignment)

        elif is_binary(formula.root):
            f1 = self.evaluate_formula(formula.first, assignment)
            f2 = self.evaluate_formula(formula.second, assignment)

            if formula.root == AND:
                return f1 and f2
            elif formula.root == OR:
                return f1 or f2
            elif formula.root == IMPLY:
                return (f1 and f2) or not f1

        elif is_quantifier(formula.root):
            if formula.root == ALL:
                for element in self.universe:
                    new_assignment = dict(assignment)
                    new_assignment[formula.variable] = element
                    # There exists one model where formula evaluates to False
                    if not self.evaluate_formula(formula.predicate, new_assignment):
                        return False
                # for all assignments of the variable of quantification,
                # formula evaluates to True
                return True
            elif formula.root == EXISTS:
                for element in self.universe:
                    new_assignment = dict(assignment)
                    new_assignment[formula.variable] = element
                    # There exists one model where formula evaluates to True
                    if self.evaluate_formula(formula.predicate, new_assignment):
                        return True
                # for all assignments of the variable of quantification,
                # formula evaluates to False
                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
        for formula in formulas:
            variables = list(formula.free_variables())
            for assignment in itertools.product(self.universe, repeat=len(variables)):
                new_assignment = {variables[i]: assignment[i] for i in range(len(variables))}
                if not self.evaluate_formula(formula, new_assignment):
                    return False
        # all formulas evaluate to true
        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

```python
# (c) This file is part of the course
# Mathematical Logic through Programming
# by Gonczarowski and Nisan.
# File name: propositions/syntax.py

"""Syntactic handling of propositional formulae."""

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

from logic_utils import frozen

EMPTY_STRING = ''
OPEN_BRACKET = '('
LEFT_BRACKET = 1
CLOSE_BRACKET = ')'
RIGHT_BRACKET = -1
UNEXPECTED_SYMBOL = 'Unexpected Symbol'


def is_variable(s: str) -> bool:
    """Checks if the given string is an atomic proposition.

    Parameters:
        s: string to check.

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


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

    Parameters:
        s: string to check.

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


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

    Parameters:
        s: string to check.

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


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

```python

    Parameters:
        s: string to check.

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


@frozen
class Formula:
    """An immutable propositional formula in tree representation.

    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) -> 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 type(first) is Formula and second is None
            self.root, self.first = root, first
        else:
            assert is_binary(root) and type(first) is Formula and \
                   type(second) is Formula
            self.root, self.first, self.second = root, first, 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
```

```
            does not equal the current formula, ``False`` otherwise.
        """
        return not self == other

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

    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
        if is_unary(self.root):
            return self.root + repr(self.first)
        if is_binary(self.root):
            return '(' + repr(self.first) + self.root + repr(self.second) + ')'

    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_constant(self.root):
            return set()
        if is_variable(self.root):
            return {self.root}
        if is_unary(self.root):
            return self.first.variables()
        if is_binary(self.root):
            return self.first.variables().union(self.second.variables())

    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()
        if is_constant(self.root):
            return {self.root}
        if is_unary(self.root):
            return {self.root}.union(self.first.operators())
        if is_binary(self.root):
            operators = self.first.operators().union(self.second.operators())
            return operators.union({self.root})

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

        Parameters:
            s: string to parse.

        Returns:
            A pair of the parsed formula and the unparsed suffix of the string.
            If the first token of the string is a variable name (e.g.,
            ``'x12'``), then the parsed prefix will be 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.
        """
```

```python
        # check if given string is empty
        if s == EMPTY_STRING:
            return None, UNEXPECTED_SYMBOL

        # Check for constant prefix
        if is_constant(s[0]):
            return Formula(s[0]), s[1:]

        # Check for variable prefix
        if is_variable(s[0]):
            index = 1
            while index < len(s) and is_variable(s[:index+1]):
                index += 1
            return Formula(s[:index]), s[index:]

        # Check for unary prefix
        if is_unary(s[0]):
            parsed = Formula.parse_prefix(s[1:])
            if parsed[1] != UNEXPECTED_SYMBOL:
                return Formula(s[0], parsed[0]), parsed[1]

        # Check for binary prefix
        if s[0] == OPEN_BRACKET:
            index = 1
            # look for the binary operator and the two formulas in its sides
            root_index_start = None
            root_index_end = None
            sum_brackets = LEFT_BRACKET
            while sum_brackets > 0 and index < len(s):
                # open bracket
                if s[index] == OPEN_BRACKET:
                    sum_brackets += LEFT_BRACKET
                # close bracket
                if s[index] == CLOSE_BRACKET:
                    sum_brackets += RIGHT_BRACKET
                # binary operation, length 1
                if is_binary(s[index]) and sum_brackets == 1:
                    if root_index_start is not None:
                        return None, UNEXPECTED_SYMBOL
                    root_index_start = index
                    root_index_end = index + 1
                # binary operation, length 2
                if (index + 1) < len(s):
                    if is_binary(s[index:index+2]) and sum_brackets == 1:
                        if root_index_start is not None:
                            return None, UNEXPECTED_SYMBOL
                        root_index_start = index
                        root_index_end = index + 2
                        index += 2
                        continue
                # binary operation, length 3
                if (index + 2) < len(s):
                    if is_binary(s[index:index + 3]) and sum_brackets == 1:
                        if root_index_start is not None:
                            return None, UNEXPECTED_SYMBOL
                        root_index_start = index
                        root_index_end = index + 3
                        index += 3
                        continue
                # else, one char was read, a bracket or a binary operation
                index += 1

            # Check if a binary operation was found
            if root_index_start is None or sum_brackets > 0:
                return None, UNEXPECTED_SYMBOL
            operation = s[root_index_start:root_index_end]
            parse_first = Formula.parse_prefix(s[1:root_index_start])
            # Check if the formula on the left side of the operation is legal
```

```python
            if parse_first[1] != EMPTY_STRING:
                return None, UNEXPECTED_SYMBOL
            # Check if the formula on the right side of the operation is legal
            parse_second = Formula.parse_prefix(s[root_index_end:index-1])
            if parse_second[1] != EMPTY_STRING:
                return None, UNEXPECTED_SYMBOL
            returned_formula =\
                Formula(operation, parse_first[0], parse_second[0])
            formula_end = s[index:]
            return returned_formula, formula_end

        # else, the prefix doesn't fit any Formula patterns
        return None, UNEXPECTED_SYMBOL


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

        Parameters:
            s: string to check.

        Returns:
            ``True`` if the given string is a valid standard string
            representation of a formula, ``False`` otherwise.
        """
        if Formula.parse_prefix(s)[1] == EMPTY_STRING:
            return True

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

        Parameters:
            s: string to parse.

        Returns:
            A formula whose standard string representation is the given string.
        """
        assert Formula.is_formula(s)
        return Formula.parse_prefix(s)[0]

    # 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.
        """
        # Optional Task 1.7

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

        Parameters:
            s: string to parse.

        Returns:
            A formula whose polish notation representation is the given string.
        """
        # Optional Task 1.8

    # Tasks for Chapter 3

    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: the mapping defining the substitutions to be
                performed.

        Returns:
            The resulting formula.

        Examples:
            >>> Formula.parse('((p->p)|z)').substitute_variables(
            ...     {'p': Formula.parse('(q&r)')})
            (((q&r)->(q&r))|z)
        """
        for variable in substitution_map:
            assert is_variable(variable)
        # root is a constant
        if is_constant(self.root):
            return self
        # root is a variable
        if is_variable(self.root):
            if self.root in substitution_map:
                return substitution_map[self.root]
            # else, root is not a variable in the mapping
            return self
        # root is an unary operation
        if is_unary(self.root):
            first = Formula.substitute_variables(self.first, substitution_map)
            return Formula(root=self.root, first=first)
        # root is an binary operation
        if is_binary(self.root):
            f1 = Formula.substitute_variables(self.first, substitution_map)
            f2 = Formula.substitute_variables(self.second, substitution_map)
            return Formula(root=self.root, first=f1, second=f2)
        # Task 3.3

    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: the mapping defining the substitutions to be
                performed.

        Returns:
            The resulting formula.

        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'})

        # root is a constant
        if is_constant(self.root):
            if self.root in substitution_map:
                return substitution_map[self.root]
            # else
            return self
```

```

        # root is a variable
        if is_variable(self.root):
            return self

        # root is an unary operation
        if is_unary(self.root):
            f1 = Formula.substitute_operators(self.first, substitution_map)
            if self.root in substitution_map:
                f = substitution_map[self.root]
                return Formula.substitute_variables(f, {'p': f1})
            # else, operator not in substitution_map
            return Formula(root=self.root, first=f1)

        # root is a binary operation
        if is_binary(self.root):
            f1 = Formula.substitute_operators(self.first, substitution_map)
            f2 = Formula.substitute_operators(self.second,
                                              substitution_map)
            if self.root in substitution_map:
                f = substitution_map[self.root]
                return Formula.substitute_variables(f, {'p': f1, 'q': f2})
            # else, operator not in substitution_map
            return Formula(root=self.root, first=f1, second=f2)
        # Task 3.4
```