import itertools

class EvaluationException(Exception):

"""Custom exception for evaluation errors in logical sentences."""

pass

class Sentence:

def evaluate(self, model):

"""Evaluates the logical sentence."""

raise Exception("Nothing to evaluate")

def formula(self):

"""Returns string formula representing logical sentence."""

return ""

def symbols(self):

"""Returns a set of all symbols in the logical sentence."""

return set()

@classmethod

def validate(cls, sentence):

if not isinstance(sentence, Sentence):

raise TypeError("Must be a logical sentence")

@classmethod

def parenthesize(cls, s):

"""Parenthesizes an expression if not already parenthesized."""

def balanced(s):

"""Checks if a string has balanced parentheses."""

count = 0

for c in s:

if c == "(":

count += 1

elif c == ")":

if count <= 0:

return False

count -= 1

return count == 0

if not len(s) or s.isalpha() or (

s[0] == "(" and s[-1] == ")" and balanced(s[1:-1])

):

return s

else:

return f"({s})"

class Symbol(Sentence):

def \_\_init\_\_(self, name):

self.name = name

def \_\_eq\_\_(self, other):

return isinstance(other, Symbol) and self.name == other.name

def \_\_hash\_\_(self):

return hash(("symbol", self.name))

def \_\_repr\_\_(self):

return self.name

def evaluate(self, model):

try:

return bool(model[self.name])

except KeyError:

raise EvaluationException(f"Variable '{self.name}' not in model")

def formula(self):

return self.name

def symbols(self):

return {self.name}

class Not(Sentence):

def \_\_init\_\_(self, operand):

Sentence.validate(operand)

self.operand = operand

def \_\_eq\_\_(self, other):

return isinstance(other, Not) and self.operand == other.operand

def \_\_hash\_\_(self):

return hash(("not", hash(self.operand)))

def \_\_repr\_\_(self):

return f"Not({self.operand})"

def evaluate(self, model):

return not self.operand.evaluate(model)

def formula(self):

return "¬" + Sentence.parenthesize(self.operand.formula())

def symbols(self):

return self.operand.symbols()

class And(Sentence):

def \_\_init\_\_(self, \*conjuncts):

for conjunct in conjuncts:

Sentence.validate(conjunct)

self.conjuncts = list(conjuncts)

def \_\_eq\_\_(self, other):

return isinstance(other, And) and self.conjuncts == other.conjuncts

def \_\_hash\_\_(self):

return hash(

("and", tuple(hash(conjunct) for conjunct in self.conjuncts))

)

def \_\_repr\_\_(self):

conjunctions = ", ".join(

[str(conjunct) for conjunct in self.conjuncts]

)

return f"And({conjunctions})"

def add(self, conjunct):

Sentence.validate(conjunct)

self.conjuncts.append(conjunct)

def evaluate(self, model):

return all(conjunct.evaluate(model) for conjunct in self.conjuncts)

def formula(self):

if len(self.conjuncts) == 1:

return self.conjuncts[0].formula()

return " ∧ ".join([Sentence.parenthesize(conjunct.formula())

for conjunct in self.conjuncts])

def symbols(self):

return set.union(\*[conjunct.symbols() for conjunct in self.conjuncts])

class Or(Sentence):

def \_\_init\_\_(self, \*disjuncts):

for disjunct in disjuncts:

Sentence.validate(disjunct)

self.disjuncts = list(disjuncts)

def \_\_eq\_\_(self, other):

return isinstance(other, Or) and self.disjuncts == other.disjuncts

def \_\_hash\_\_(self):

return hash(

("or", tuple(hash(disjunct) for disjunct in self.disjuncts))

)

def \_\_repr\_\_(self):

disjuncts = ", ".join([str(disjunct) for disjunct in self.disjuncts])

return f"Or({disjuncts})"

def evaluate(self, model):

return any(disjunct.evaluate(model) for disjunct in self.disjuncts)

def formula(self):

if len(self.disjuncts) == 1:

return self.disjuncts[0].formula()

return " ∨ ".join([Sentence.parenthesize(disjunct.formula())

for disjunct in self.disjuncts])

def symbols(self):

return set.union(\*[disjunct.symbols() for disjunct in self.disjuncts])

class Implication(Sentence):

def \_\_init\_\_(self, antecedent, consequent):

Sentence.validate(antecedent)

Sentence.validate(consequent)

self.antecedent = antecedent

self.consequent = consequent

def \_\_eq\_\_(self, other):

return (isinstance(other, Implication)

and self.antecedent == other.antecedent

and self.consequent == other.consequent)

def \_\_hash\_\_(self):

return hash(("implies", hash(self.antecedent), hash(self.consequent)))

def \_\_repr\_\_(self):

return f"Implication({self.antecedent}, {self.consequent})"

def evaluate(self, model):

return ((not self.antecedent.evaluate(model))

or self.consequent.evaluate(model))

def formula(self):

antecedent = Sentence.parenthesize(self.antecedent.formula())

consequent = Sentence.parenthesize(self.consequent.formula())

return f"{antecedent} => {consequent}"

def symbols(self):

return set.union(self.antecedent.symbols(), self.consequent.symbols())

class Biconditional(Sentence):

def \_\_init\_\_(self, left, right):

Sentence.validate(left)

Sentence.validate(right)

self.left = left

self.right = right

def \_\_eq\_\_(self, other):

return (isinstance(other, Biconditional)

and self.left == other.left

and self.right == other.right)

def \_\_hash\_\_(self):

return hash(("biconditional", hash(self.left), hash(self.right)))

def \_\_repr\_\_(self):

return f"Biconditional({self.left}, {self.right})"

def evaluate(self, model):

return ((self.left.evaluate(model)

and self.right.evaluate(model))

or (not self.left.evaluate(model)

and not self.right.evaluate(model)))

def formula(self):

left = Sentence.parenthesize(self.left.formula())

right = Sentence.parenthesize(self.right.formula())

return f"{left} <=> {right}"

def symbols(self):

return set.union(self.left.symbols(), self.right.symbols())

def model\_check(knowledge, query):

"""Checks if knowledge base entails query."""

def check\_all(knowledge, query, symbols, model):

"""Recursively checks all possible models."""

# If model has an assignment for each symbol

if not symbols:

# If knowledge base is true in model, then query must also be true

if knowledge.evaluate(model):

return query.evaluate(model)

return True

else:

# Choose one of the remaining unused symbols

remaining = symbols.copy()

p = remaining.pop()

# Create a model where the symbol is true

model\_true = model.copy()

model\_true[p] = True

# Create a model where the symbol is false

model\_false = model.copy()

model\_false[p] = False

# Ensure entailment holds in both models

return (check\_all(knowledge, query, remaining, model\_true) and

check\_all(knowledge, query, remaining, model\_false))

# Get all symbols in both knowledge and query

symbols = set.union(knowledge.symbols(), query.symbols())

# Check that knowledge entails query

return check\_all(knowledge, query, symbols, dict())

if \_\_name\_\_ == "\_\_main\_\_":

# Example usage

# Define symbols

rain = Symbol('Rain')

traffic = Symbol('Traffic')

carry\_umbrella = Symbol('Carry\_Umbrella')

have\_good\_day = Symbol('Have\_Good\_Day')

# Define knowledge base

# If it rains, then carry an umbrella

knowledge1 = Implication(rain, carry\_umbrella)

# If there's traffic, then have a bad day

knowledge2 = Implication(traffic, Not(have\_good\_day))

# If you carry an umbrella, then have a good day

knowledge3 = Implication(carry\_umbrella, have\_good\_day)

# Combine all knowledge

knowledge = And(knowledge1, knowledge2, knowledge3)

# Define query

query = have\_good\_day

# Perform model checking

entails = model\_check(knowledge, query)

print(f"Knowledge entails query: {entails}")