ARTIFICIAL INTELLIGENCE

RA1811030010023

Sayan Pramanick

LAB – 7

AIM: Unification and Resolution

CODE

Unification:

```
class Substitution:
  def __init__(self, variable, replacement):
    self.variable = variable
    self.replacement = replacement

  def __str__(self):
    return str(self.variable) + " = " + str(self.replacement)


class Variable:
    def __init__(self, variable_name):
        if variable_name[0].islower(): raise (Exception("Variable name
starting with lower-case!"))
        self.variable_name = variable_name

    def __eq__(self, other):
        return isinstance(other, Variable) and self.variable_name == ot
her.variable_name

    def __ne__(self, other):
        return not self.__eq__(other)

    def __str__(self):
        return self.variable_name

    def __repr__(self):
        return str(self)

    def __hash__(self):
        return str(self).__hash__()
```

```
    def occurs_in(self, other):
        if isinstance(other, Variable) and self.__eq__(other):
            return True
        if isinstance(other, Expression) and self.__str__() in other.__
str__():
            return True
        return False


class Constant:
    def __init__(self, constant_name):
        if constant_name[0].isupper(): raise (Exception("Constant name
starting with upper-case!"))
        self.constant_name = constant_name

    def __eq__(self, other):
        return isinstance(other, Constant) and self.constant_name == ot
her.constant_name

    def __ne__(self, other):
        return not self.__eq__(other)

    def __str__(self):
        return self.constant_name

    def __repr__(self):
        return str(self)

    def __hash__(self):
        return str(self).__hash__()


class Expression:

    def __init__(self, operator, arguments):
        self.operator = operator
        self.arguments = arguments

    def __str__(self):
        return "%s(%s)" % (
            self.operator,
            ", ".join(map(str, self.arguments)))

    def __repr__(self):
        return str(self)

    def __hash__(self):
        return str(self).__hash__()
```

```
    def __eq__(self, other):
        if not isinstance(other, Expression): return False
        if self.operator != other.operator: return False
        if len(self.arguments) != len(other.arguments): return False
        return all([a1 == a2 for a1, a2 in zip(self.arguments, other.ar
guments)])

    def __ne__(self, other):
        return not self.__eq__(other)


def parse_expression(s):
    l, d, i = [], 0, 0
    op, args = None, []
    for j, c in enumerate(s):
        if c == "(":
            if op is None:
                op = s[:j]
                i = j + 1
            d += 1
        if c == ")":
            if d == 1:
                if j > i: args.append(s[i:j])
                i = j + 1
            d -= 1
        if c == "," and d == 1:
            args.append(s[i:j])
            i = j + 1
        if c == " " and i == j: i += 1

    if op is None:
        if s[0].isupper():
            return Variable(s)
        else:
            return Constant(s)

    return Expression(op, list(map(parse_expression, args)))

def unify_with_occurrence_check(formular1, formular2, mgu = [], trace =
 False):
  #pp(trace, "Unifying expression:", formular1, "with expression:", for
mular2)
  if mgu is None:
    return None
  elif formular1 == formular2:
    return mgu
  elif isinstance(formular1, Variable):
    return unify_variable(formular1, formular2, mgu, trace)
```

```
  elif isinstance(formular2, Variable):
    return unify_variable(formular2, formular1, mgu, trace)
  elif isinstance(formular1, Expression) and isinstance(formular2, Expr
ession):
    if type(formular1) != type(formular2) or formular1.operator != form
ular2.operator or len(formular1.arguments) != len(formular2.arguments):
      return None
    else:
      for a,b in zip(formular1.arguments, formular2.arguments):
        mgu = unify_with_occurrence_check(a, b, mgu, trace)
      return mgu
  else:
    return None



def substitute(sub, expr):
  for s in (x for x in sub if occurs_in(x.variable, expr)):
    if isinstance(expr, Variable):
      expr = s.replacement
    else:
      expr.arguments = [substitute(sub, e) for e in expr.arguments]
  return expr



def occurs_in(var, expr):
  if var == expr:
    return True
  if not isinstance(expr, Expression):
    return False
  return any([occurs_in(var, e) for e in expr.arguments])



def unify_variable(var, exp, mgu, trace):
  for s in (x for x in mgu if x.variable == var):
    return unify_with_occurrence_check(s.replacement, exp, mgu, trace)
  t = substitute(mgu, exp)
  if occurs_in(var, t) and isinstance(t, Expression):
    print("\nCannot unify - infinte loop exception!!!")
    return None
  else:
    s = Substitution(var, t)
    mgu = mgu+[s]
    for q in (x for x in mgu if x.replacement == s.variable):
      mgu.remove(q)
      new  = Substitution(q.variable, s.replacement)
```

```
      mgu = mgu+[new]
    for r in (x for x in mgu if isinstance(x.replacement, Expression)):
      mgu.remove(r)
      a = substitute(mgu, r.replacement)
      b = Substitution(r.variable, a)
      mgu = mgu+[b]
    for s in (x for x in mgu if (x.variable == x.replacement)):
      #print("Variable already unified, duplicate deleted!!!")
      mgu.remove(s)
    return mgu


def main():

  keep_running = True

  while keep_running:

    print("\nPlease enter the first term:")
    t1 = input("-->")
    print("\nPlease enter the second term:")
    t2 = input("-->")

    mgu = unify_with_occurrence_check(parse_expression(t1), parse_expre
ssion(t2), trace = False)
    if mgu is None:
      print("\nno")
    else:
      print("\n")
      print("\n".join(map(str, mgu)))
      print("\nyes")


    print("\n\n>>> Do you want to run unifier again? (Y/N)")
    re_run = input("--> ")

    if re_run != "y" and re_run != "Y":
      keep_running = False

if __name__ == "__main__":
  main()
```

Output:

```
Please enter the first term:
-->P(B,A)

Please enter the second term:
-->P(A,B)


B = A

yes


>>> Do you want to run unifier again? (Y/N)
--> n
```

Resolution :

```
import copy

import time


class Parameter:
   variable_count = 1


   def _init_(self, name=None):
      if name:
         self.type = "Constant"
         self.name = name
      else:
         self.type = "Variable"
         self.name = "v" + str(Parameter.variable_count)
         Parameter.variable_count += 1
```

```
    def isConstant(self):
        return self.type == "Constant"

    def unify(self, type_, name):
        self.type = type_
        self.name = name

    def _eq_(self, other):
        return self.name == other.name

    def _str_(self):
        return self.name

class Predicate:
    def _init_(self, name, params):
        self.name = name
        self.params = params

    def _eq_(self, other):
        return self.name == other.name and all(a == b for a, b in
zip(self.params, other.params))

    def _str_(self):
```

```
        return self.name + "(" + ",".join(str(x) for x in self.params) + ")"

    def getNegatedPredicate(self):
        return Predicate(negatePredicate(self.name), self.params)


class Sentence:
    sentence_count = 0

    def _init_(self, string):
        self.sentence_index = Sentence.sentence_count
        Sentence.sentence_count += 1
        self.predicates = []
        self.variable_map = {}
        local = {}

        for predicate in string.split("|"):
            name = predicate[:predicate.find("(")]
            params = []

            for param in predicate[predicate.find("(") + 1:
predicate.find(")")].split(","):
                if param[0].islower():
                    if param not in local:  # Variable
```

```
                local[param] = Parameter()
                self.variable_map[local[param].name] = local[param]
            new_param = local[param]
        else:
            new_param = Parameter(param)
            self.variable_map[param] = new_param

        params.append(new_param)

    self.predicates.append(Predicate(name, params))

def getPredicates(self):
    return [predicate.name for predicate in self.predicates]

def findPredicates(self, name):
    return [predicate for predicate in self.predicates if
predicate.name == name]

def removePredicate(self, predicate):
    self.predicates.remove(predicate)
    for key, val in self.variable_map.items():
        if not val:
            self.variable_map.pop(key)
```

```
    def containsVariable(self):
        return any(not param.isConstant() for param in
self.variable_map.values())

    def _eq_(self, other):
        if len(self.predicates) == 1 and self.predicates[0] == other:
            return True
        return False

    def _str_(self):
        return "".join([str(predicate) for predicate in self.predicates])

class KB:
    def _init_(self, inputSentences):
        self.inputSentences = [x.replace(" ", "") for x in inputSentences]
        self.sentences = []
        self.sentence_map = {}

    def prepareKB(self):
        self.convertSentencesToCNF()
        for sentence_string in self.inputSentences:
            sentence = Sentence(sentence_string)
            for predicate in sentence.getPredicates():
```

```
            self.sentence_map[predicate] =
self.sentence_map.get(predicate, []) + [sentence]


    def convertSentencesToCNF(self):
        for sentenceIdx in range(len(self.inputSentences)):
            if "=>" in self.inputSentences[sentenceIdx]:  # Do negation of
the Premise and add them as literal
                self.inputSentences[sentenceIdx] =
negateAntecedent(self.inputSentences[sentenceIdx])


    def askQueries(self, queryList):
        results = []


        for query in queryList:
            negatedQuery = Sentence(negatePredicate(query.replace(" ",
"")))
            negatedPredicate = negatedQuery.predicates[0]
            prev_sentence_map = copy.deepcopy(self.sentence_map)
            self.sentence_map[negatedPredicate.name] =
self.sentence_map.get(negatedPredicate.name, []) + [negatedQuery]
            self.timeLimit = time.time() + 40


            try:
                result = self.resolve([negatedPredicate],
[False]*(len(self.inputSentences) + 1))
```

```
            except:
                result = False

            self.sentence_map = prev_sentence_map

            if result:
                results.append("TRUE")
            else:
                results.append("FALSE")

        return results

    def resolve(self, queryStack, visited, depth=0):
        if time.time() > self.timeLimit:
            raise Exception
        if queryStack:
            query = queryStack.pop(-1)
            negatedQuery = query.getNegatedPredicate()
            queryPredicateName = negatedQuery.name
            if queryPredicateName not in self.sentence_map:
                return False
            else:
                queryPredicate = negatedQuery
```

```
            for kb_sentence in
self.sentence_map[queryPredicateName]:
                if not visited[kb_sentence.sentence_index]:
                    for kbPredicate in
kb_sentence.findPredicates(queryPredicateName):

                        canUnify, substitution =
performUnification(copy.deepcopy(queryPredicate),
copy.deepcopy(kbPredicate))

                        if canUnify:
                            newSentence = copy.deepcopy(kb_sentence)
                            newSentence.removePredicate(kbPredicate)
                            newQueryStack = copy.deepcopy(queryStack)

                            if substitution:
                                for old, new in substitution.items():
                                    if old in newSentence.variable_map:
                                        parameter =
newSentence.variable_map[old]
                                        newSentence.variable_map.pop(old)
                                        parameter.unify("Variable" if
new[0].islower() else "Constant", new)
                                        newSentence.variable_map[new] =
parameter
```

```
                        for predicate in newQueryStack:
                            for index, param in
enumerate(predicate.params):
                                if param.name in substitution:
                                    new = substitution[param.name]
                                    predicate.params[index].unify("Variable"
if new[0].islower() else "Constant", new)

                        for predicate in newSentence.predicates:
                            newQueryStack.append(predicate)

                        new_visited = copy.deepcopy(visited)
                        if kb_sentence.containsVariable() and
len(kb_sentence.predicates) > 1:
                            new_visited[kb_sentence.sentence_index] =
True

                        if self.resolve(newQueryStack, new_visited, depth
+ 1):
                            return True
                return False
        return True


def performUnification(queryPredicate, kbPredicate):
```

```
    substitution = {}
    if queryPredicate == kbPredicate:
        return True, {}
    else:
        for query, kb in zip(queryPredicate.params,
kbPredicate.params):
            if query == kb:
                continue
            if kb.isConstant():
                if not query.isConstant():
                    if query.name not in substitution:
                        substitution[query.name] = kb.name
                    elif substitution[query.name] != kb.name:
                        return False, {}
                    query.unify("Constant", kb.name)
                else:
                    return False, {}
            else:
                if not query.isConstant():
                    if kb.name not in substitution:
                        substitution[kb.name] = query.name
                    elif substitution[kb.name] != query.name:
                        return False, {}
                    kb.unify("Variable", query.name)
```

```
            else:
                if kb.name not in substitution:
                    substitution[kb.name] = query.name
                elif substitution[kb.name] != query.name:
                    return False, {}
    return True, substitution


def negatePredicate(predicate):
    return predicate[1:] if predicate[0] == "" else "" + predicate


def negateAntecedent(sentence):
    antecedent = sentence[:sentence.find("=>")]
    premise = []

    for predicate in antecedent.split("&"):
        premise.append(negatePredicate(predicate))

    premise.append(sentence[sentence.find("=>") + 2:])
    return "|".join(premise)


def getInput(filename):
```

```
    with open(filename, "r") as file:
        noOfQueries = int(file.readline().strip())
        inputQueries = [file.readline().strip() for _ in
range(noOfQueries)]
        noOfSentences = int(file.readline().strip())
        inputSentences = [file.readline().strip() for _ in
range(noOfSentences)]
        return inputQueries, inputSentences



def printOutput(filename, results):
    print(results)
    with open(filename, "w") as file:
        for line in results:
            file.write(line)
            file.write("\n")
    file.close()



if _name_ == '_main_':
    inputQueries_, inputSentences_ =
getInput("/home/ubuntu/environment/RA1811003010435/input1.tx
t")
    knowledgeBase = KB(inputSentences_)
    knowledgeBase.prepareKB()
```

```
    results_ = knowledgeBase.askQueries(inputQueries_)
    printOutput("output.txt", results_)
```

Input:

```
6
F(Joe)
H(John)
~H(Alice)
~H(John)
G(Joe)
G(Tom)
14
~F(x) | G(x)
~G(x) | H(x)
~H(x) | F(x)
~R(x) | H(x)
~A(x) | H(x)
~D(x,y) | ~H(y)
~B(x,y) | ~C(x,y) | A(x)
B(John,Alice)
B(John,Joe)
~D(x,y) | ~Q(y) | C(x,y)
D(John,Alice)
Q(Joe)
D(John,Joe)
```

R(Tom)

Output:

input.txt ouput.txt

```
1 FALSE
2 TRUE
3 TRUE
4 FALSE
5 FALSE
6 TRUE
7
```