A N V SREEVISHNU

RA1811003010333

EXP 7: Unification and Resolution

Unification

Aim:

Implementation of unification for real world problems.

Algorithm:

```
Step 1: If Ψ1 or Ψ2 is a variable or constant, then:
        a, If Ψ1 or Ψ2 are identical, then return NULL.
        b, Else if Ψ1 is a variable:
            - then if Ψ1 occurs in Ψ2, then return False
            - Else return (Ψ2 / Ψ1)
        c, Else if Ψ2 is a variable:
            - then if Ψ2 occurs in Ψ1, then return False
            - Else return (Ψ1 / Ψ2)
        d, Else return False
Step 2: If the initial Predicate symbol in Ψ1 and Ψ2 are not same, then return False.
Step 3: IF Ψ1 and Ψ2 have a different number of arguments, then return False.
Step 4: Create Substitution list.
Step 5: For i=1 to the number of elements in Ψ1.
        a, Call Unify function with the ith element of Ψ1 and ith element of Ψ2, and put the result into S.
        b, If S = False then returns False
        c, If S ≠ Null then append to Substitution list
 Step 6: Return Substitution list.
```

Code:

```
def get_index_comma(string):
    """
    Return index of commas in string
    """

    index_list = list()
    # Count open parentheses
    par_count = 0

    for i in range(len(string)):
        if string[i] == ',' and par_count == 0:
            index_list.append(i)
        elif string[i] == '(':
            par_count += 1
        elif string[i] == ')':
            par_count -= 1

    return index_list



def is_variable(expr):
    """
    Check if expression is variable
    """
    for i in expr:
        if i == '(':
            return False

    return True


def process_expression(expr):
    """
    input:  - expression:
            'Q(a, g(x, b), f(y))'
    return: - predicate symbol:
            Q
            - list of arguments
            ['a', 'g(x, b)', 'f(y)']
    """

    # Remove space in expression
    expr = expr.replace(' ', '')

    # Find the first index == '('
    index = None
    for i in range(len(expr)):
        if expr[i] == '(':
            index = i
            break

    # Return predicate symbol and remove predicate symbol in expression
    predicate_symbol = expr[:index]
    expr = expr.replace(predicate_symbol, '')

    # Remove '(' in the first index and ')' in the last index
    expr = expr[1:len(expr) - 1]

    # List of arguments
    arg_list = list()

    # Split string with commas, return list of arguments
    indices = get_index_comma(expr)

    if len(indices) == 0:
        arg_list.append(expr)
    else:
        arg_list.append(expr[:indices[0]])
        for i, j in zip(indices, indices[1:]):
            arg_list.append(expr[i + 1:j])
        arg_list.append(expr[indices[len(indices) - 1] + 1:])

    return predicate_symbol, arg_list


def get_arg_list(expr):
    """
    input:  expression:
            'Q(a, g(x, b), f(y))'
    return: full list of arguments:
            ['a', 'x', 'b', 'y']
    """
    _, arg_list = process_expression(expr)

    flag = True
    while flag:
        flag = False

        for i in arg_list:
            if not is_variable(i):
                flag = True
                _, tmp = process_expression(i)
                for j in tmp:
                    if j not in arg_list:
                        arg_list.append(j)
                arg_list.remove(i)

    return arg_list


def check_occurs(var, expr):
    """
    Check if var occurs in expr
    """

    arg_list = get_arg_list(expr)
    if var in arg_list:
        return True

    return False
```

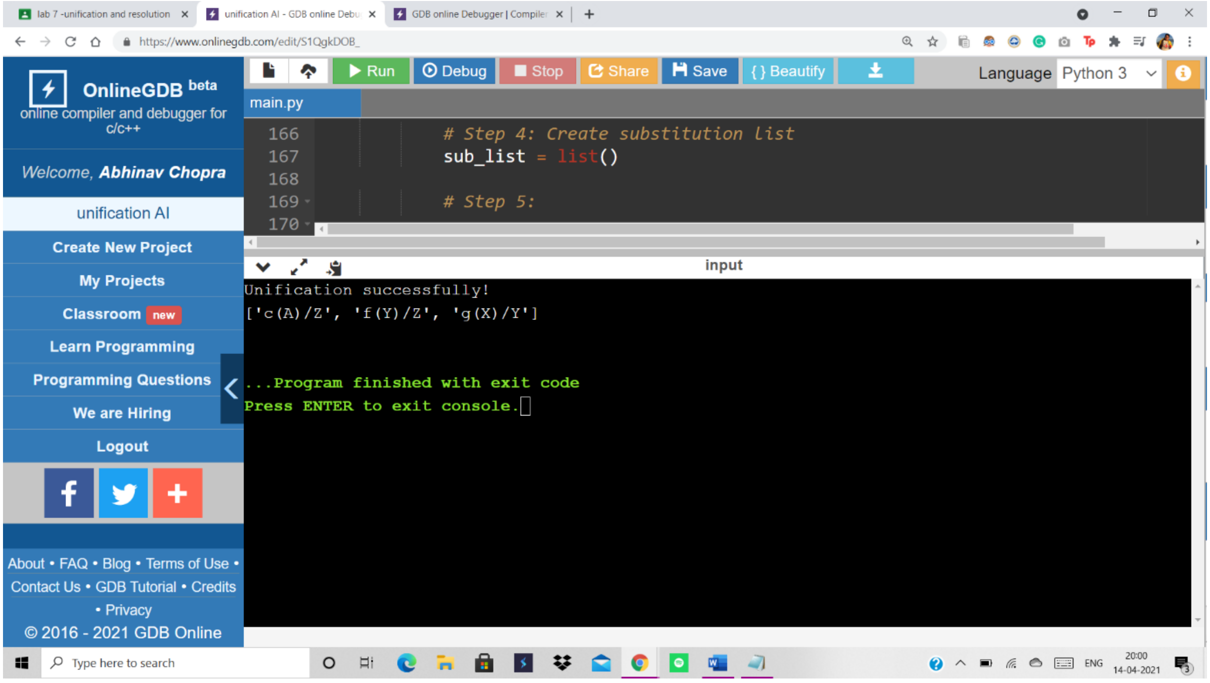
```
def unify(expr1, expr2):
    # Step 1:
    if is_variable(expr1) and is_variable(expr2):
        if expr1 == expr2:
            return 'Null'
        else:
            return False
    elif is_variable(expr1) and not is_variable(expr2):
        if check_occurs(expr1, expr2):
            return False
        else:
            tmp = str(expr2) + '/' + str(expr1)
            return tmp
    elif not is_variable(expr1) and is_variable(expr2):
        if check_occurs(expr2, expr1):
            return False
        else:
            tmp = str(expr1) + '/' + str(expr2)
            return tmp
    else:
        predicate_symbol_1, arg_list_1 = process_expression(expr1)
        predicate_symbol_2, arg_list_2 = process_expression(expr2)

        # Step 2
        if predicate_symbol_1 != predicate_symbol_2:
            return False
        # Step 3
        elif len(arg_list_1) != len(arg_list_2):
            return False
        else:
            # Step 4: Create substitution list
            sub_list = list()

            # Step 5:
            for i in range(len(arg_list_1)):
                tmp = unify(arg_list_1[i], arg_list_2[i])

                if not tmp:
                    return False
                elif tmp == 'Null':
                    pass
                else:
                    if type(tmp) == list:
                        for j in tmp:
                            sub_list.append(j)
                    else:
                        sub_list.append(tmp)

            # Step 6
            return sub_list


if __name__ == '__main__':
    # Data 1
    f1 = 'p(c(A), Z, f(g(X)))'
    f2 = 'p(Z, f(Y), f(Y))'
```

```
result = unify(f1, f2)
if not result:
    print('Unification failed!')
else:
    print('Unification successfully!')
    print(result)
```

Output:

Unification successfully!

['c(A)/Z', 'f(Y)/Z', 'g(X)/Y']



```
    # Step 4: Create substitution list
    sub_list = list()

    # Step 5:
```

```
Unification successfully!
['c(A)/Z', 'f(Y)/Z', 'g(X)/Y']


...Program finished with exit code
Press ENTER to exit console.
```

Result:

Thus, the implementation of unification for real world problems is successfully executed using python.

Resolution:

Aim:

Implementation of resolution for real world problems.

Algorithm:

- Conversion of facts to first order logic
- Convert FOL statements to CNF
- Negate the statements which is to be proved (proof by contradiction)
- Draw resolution graph(unification)
- Exit

Code:

```
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
```

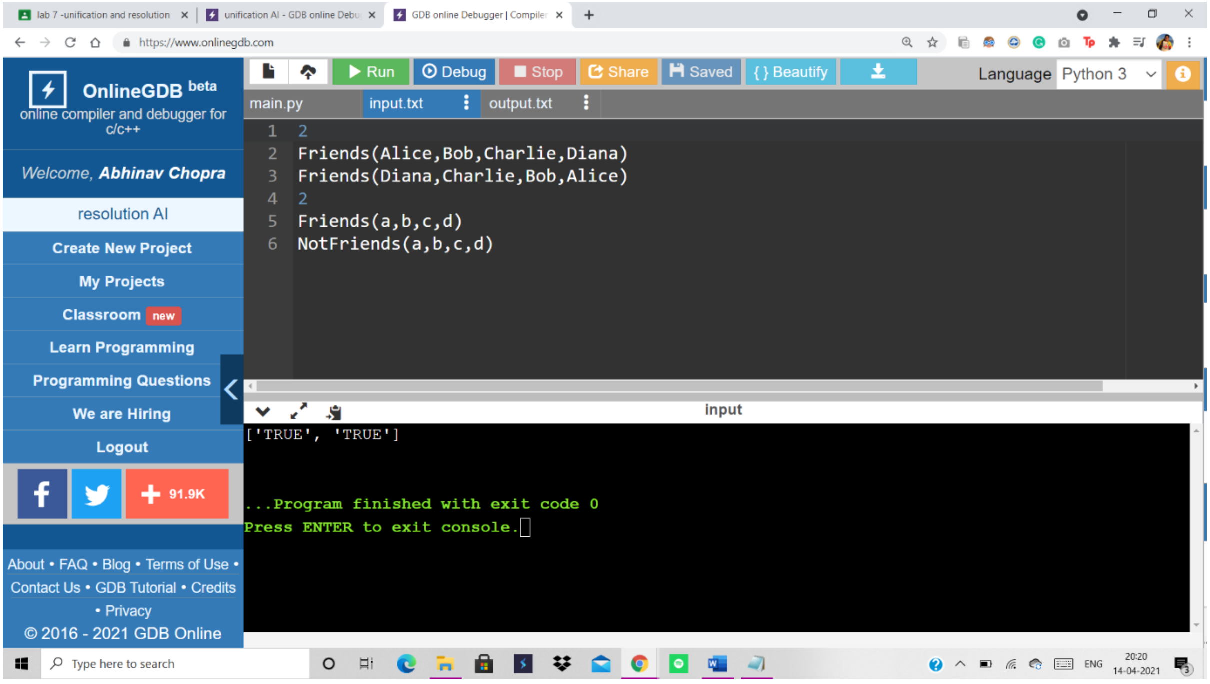
```
def __init__(self, string):
   self.sentence_index = Sentence.sentence_count
   Sentence.sentence_count += 1
   self.predicates = []
   self.variable_map = {}
   local = {}

   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("input.txt")
    knowledgeBase = KB(inputSentences_)
    knowledgeBase.prepareKB()
    results_ = knowledgeBase.askQueries(inputQueries_)
    printOutput("output.txt", results_)
```

Output:

['TRUE', 'TRUE']



```
2
Friends(Alice,Bob,Charlie,Diana)
Friends(Diana,Charlie,Bob,Alice)
2
Friends(a,b,c,d)
NotFriends(a,b,c,d)
```

```
['TRUE', 'TRUE']


...Program finished with exit code 0
Press ENTER to exit console.
```

Result:

Thus, the implementation of resolution for real world problems is successfully executed using python.