Ex No.7: *Implementation of Unification and Resolution Algorithm*

Unification

In logic and computer science, unification is a process of automatically solving equations between symbolic terms. Unification has several interesting applications, notably in logic programming. Unification is just like pattern matching, except that both terms can contain variables. So, we can no longer say one is the pattern term and the other the constant term. For example:

- First term: f(a, V, bar(D))
- Second term f(D, k, bar(a))

Given two such terms, finding a variable substitution that will make them equivalent is called unification. In this case the substitution is {D=a, V=k}.

Note that there is an infinite number of possible unifiers for some solvable unification problem. For example, given:

- First term: f(X, Y)
- Second term: f(Z, g(X))

We have the substitution {X=Z, Y=g(X)} but also something like {X=K, Z=K, Y=g(K)} and {X=j(K), Z=j(K), Y=g(j(K))} and so on. The first substitution is the simplest one, and also the most general. It's called the most general unifier or most general unifier (MGU) . Intuitively, the most general unifier (MGU) can be turned into any other unifier by performing another substitution. For example {X=Z, Y=g(X)} can be turned into {X=j(K), Z=j(K), Y=g(j(K))} by applying the substitution {Z=j(K)} to it. Note that the reverse doesn't work, as we can't turn the second into the first by using a substitution. So, we say that {X=Z, Y=g(X)} is the most general unifier for the two given terms, and it's the most general unifier (MGU) we want to find.

Algorithm :

```
1: procedure Unify(t1,t2)
2:     Inputs
3:         t1,t2: atoms or terms
4:     Output
5:         most general unifier of t1 and t2 if it exists or ⊥ otherwise
6:     Local
7:         E: a set of equality statements
8:         S: substitution
9:     E←{t1=t2}
10:     S={}
11:     while E≠{} do
12:         select and remove α=β from E
13:         if β is not identical to α then
14:             if α is a variable then
15:                 replace α with β everywhere in E and S
16:                 S←  {α/β}∪S
17:             else if β is a variable then
18:                 replace β with α everywhere in E and S
19:                 S←  {β/α}∪S
20:             else if α is p  (α1,…,αn) and β is p  (β1,…,βn) then
21:                 E←E∪{α1=β1,…,αn=βn}
22:             else
23:                 return ⊥
24:     return S
```

Code :

```
def get_index_comma(string):
    index_list = list()
    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
```

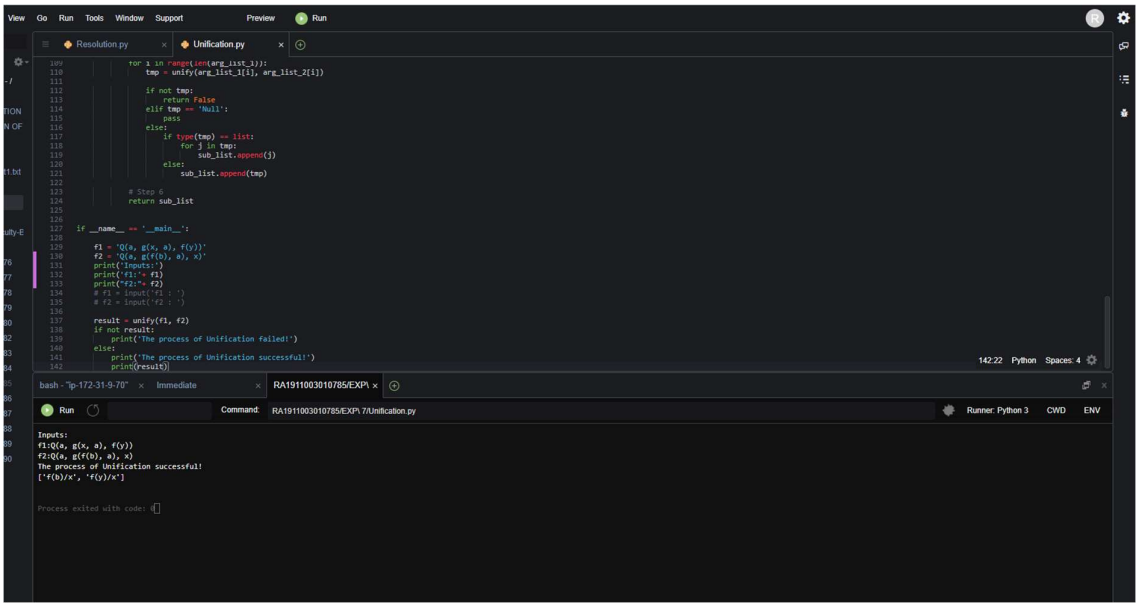
```
        elif string[i] == ')':
            par_count -= 1

    return index_list


def is_variable(expr):
    for i in expr:
        if i == '(' or i == ')':
            return False

    return True


def process_expression(expr):
    expr = expr.replace(' ', '')
    index = None
    for i in range(len(expr)):
        if expr[i] == '(':
            index = i
            break
    predicate_symbol = expr[:index]
    expr = expr.replace(predicate_symbol, '')
    expr = expr[1:len(expr) - 1]
    arg_list = list()
    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):
    _, 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):
    arg_list = get_arg_list(expr)
    if var in arg_list:
        return True

    return False


def unify(expr1, expr2):

    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__':

    f1 = 'Q(a, g(x, a), f(y))'
    f2 = 'Q(a, g(f(b), a), x)'
    print('Inputs:')
    print('f1:'+ f1)
    print("f2:"+ f2)
    # f1 = input('f1 : ')
    # f2 = input('f2 : ')
```

```
result = unify(f1, f2)
if not result:
    print('The process of Unification failed!')
else:
    print('The process of Unification successful!')
    print(result)
```

Output:



Resolution

Resolution method is an inference rule which is used in both Propositional as well as First-order Predicate Logic in different ways. This method is basically used for proving the satisfiability of a sentence. In resolution method, we use Proof by Refutation technique to prove the given statement.
The key idea for the resolution method is to use the knowledge base and negated goal to obtain null clause (which indicates contradiction). Resolution method is also called Proof by Refutation. Since the knowledge base itself is consistent, the contradiction must be introduced by a negated goal. As a result, we have to conclude that the original goal is true.

Algorithm :

1. Convert the given axiom into clausal form, i.e., disjunction form.
2. Apply and proof the given goal using negation rule.
3. Use those literals which are needed to prove.
4. Solve the clauses together and achieve the goal.

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)):
            # Do negation of the Premise and add them as literal
            if "=>" in self.inputSentences[sentenceIdx]:
```

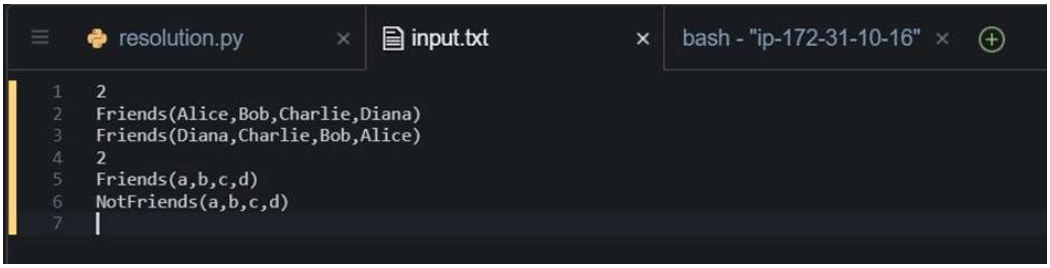
```
            self.inputSentences[sentenceIdx] = negateAntecedent(
                self.inputSentences[sentenceIdx])

    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('C:/shushrut/studies/SRM
    University/SEM 6/AI/7-Unification Resolutiion/Resolution/Input/input_1.txt')
        knowledgeBase = KB(inputSentences_)
        knowledgeBase.prepareKB()
        results_ = knowledgeBase.askQueries(inputQueries_)
        printOutput("output.txt", results_)
```

Output:



Result :

Unification and resolution were implemented successfully.