**Artificial Intelligence Lab**

**Name** - N Aditya Sai

**Register Number**- RA1911030010075

**Topic**: Unification

**Experiment** – 7a

**Aim:** To study and implement unification

**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)'

# f1 = input('f1 : ')

# f2 = input('f2 : ')

result = unify(f1, f2)

if not result:

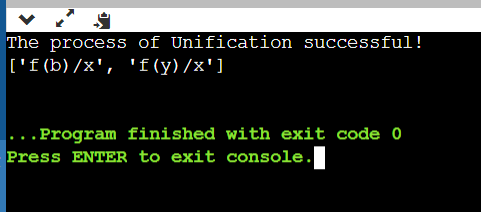
print('The process of Unification failed!')

else:

print('The process of Unification successful!')

print(result)

**Output:**



**Result:** Unification was achieved successfully.

**Artificial Intelligence Lab**

**Name** - N Aditya Sai

**Register Number**- RA1911030010075

**Topic**: Resolution

**Experiment** – 7b

**Aim:** To study and implement resolution

**Code:**

import copy

import time

class Parameter:

 variable\_count = 1

 def \_\_init\_\_(self, name=None):

 if name:

 self.type = &quot;Constant&quot;

 self.name = name

 else:

 self.type = &quot;Variable&quot;

 self.name = &quot;v&quot; +

str(Parameter.variable\_count)

 Parameter.variable\_count += 1

 def isConstant(self):

 return self.type == &quot;Constant&quot;

 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 + &quot;(&quot; + &quot;,&quot;.join(str(x)

for x in self.params) + &quot;)&quot; 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(&quot;|&quot;):

 name = predicate[:predicate.find(&quot;(&quot;)]

 params = []

 for param in predicate[predicate.find(&quot;(&quot;) + 1:

predicate.find(&quot;)&quot;)].split(&quot;,&quot;):

 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 &quot;&quot;.join([str(predicate) for predicate in

self.predicates])

class KB:

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

 self.inputSentences = [x.replace(&quot; &quot;, &quot;&quot;) 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 &quot;=&gt;&quot; 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(&quot; &quot;, &quot;&quot;)))

 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(&quot;TRUE&quot;)

 else:

 results.append(&quot;FALSE&quot;)

 return results def resolve(self, queryStack, visited, depth=0):

 if time.time() &gt; 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(&quot;Variable&quot; if new[0].islower() else

&quot;Constant&quot;, 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(&quot;Variable&quot; if

new[0].islower() else &quot;Constant&quot;, new) for predicate in

newSentence.predicates:

newQueryStack.append(predicate)

 new\_visited =

copy.deepcopy(visited)

 if kb\_sentence.containsVariable() and

len(kb\_sentence.predicates) &gt; 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(&quot;Constant&quot;, 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(&quot;Variable&quot;, query.name)

 else:

 if kb.name not in substitution:

 substitution[kb.name] = query.name

pg. 10

 elif substitution[kb.name] !=

query.name:

 return False, {}

 return True, substitutiondef negatePredicate(predicate):

 return predicate[1:] if predicate[0] == &quot;~&quot; else &quot;~&quot; +

predicate

def negateAntecedent(sentence):

 antecedent = sentence[:sentence.find(&quot;=&gt;&quot;)]

 premise = []

 for predicate in antecedent.split(&quot;&amp;&quot;):

 premise.append(negatePredicate(predicate))

 premise.append(sentence[sentence.find(&quot;=&gt;&quot;) + 2:])

 return &quot;|&quot;.join(premise)

def getInput(filename):

 with open(filename, &quot;r&quot;) 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, &quot;w&quot;) as file:

 for line in results:

 file.write(line)

 file.write(&quot;\n&quot;)

 file.close()

if \_\_name\_\_ == &#39;\_\_main\_\_&#39;:

 inputQueries\_, inputSentences\_ = getInput(&quot;input.txt&quot;)

 knowledgeBase = KB(inputSentences\_)

 knowledgeBase.prepareKB()

 results\_ = knowledgeBase.askQueries(inputQueries\_)

 printOutput(&quot;output.txt&quot;, results\_)

**Output:**

****

**Result:** Resolution was implemented.