# EXPERIMENT 7

# a) UNIFICATION ALGORITHM

**NAME:** Rahul Goel

**REG NO:** RA1911030010094

**AIM**: To implement unification algorithm .

**PROCEDURE**:

1) 2)

Initialize the substitution set to be empty.

Recursively unify atomic sentences:

- Check for Identical expression match.

- If one expression is a variable $v_i$, and the other is a term $t_i$ which does not
  contain variable $v_i$, then:

- Substitute $t_i$ / $v_i$ in the existing substitutions

- Add $t_i$ /$v_i$ to the substitution setlist.

- If both the expressions are functions, then function name must be similar, and
  the number of arguments must be the same in both the expression.

**For each pair of the following atomic sentences find the most general unifier (If exist).**

**INPUT 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
```

pg. 4

pg. 5

```
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 ':

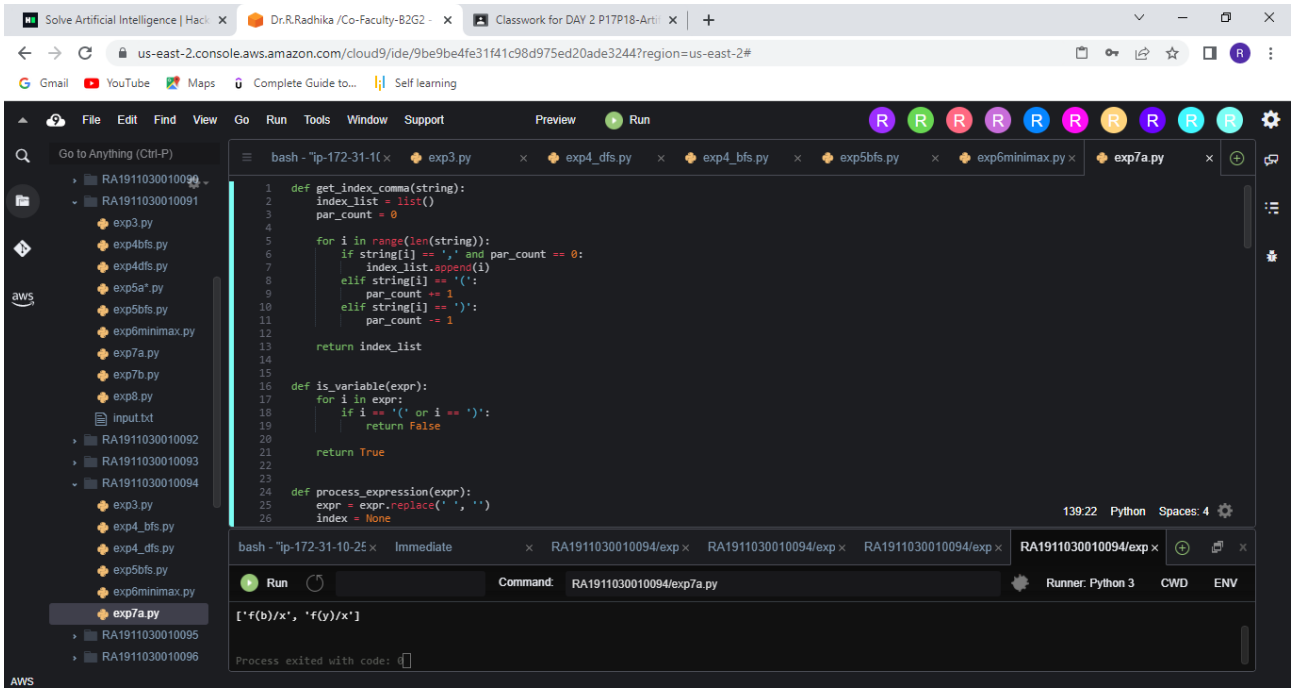
f1 = 'Q(a, g(x, a), f(y))' f2 = 'Q(a, g(f(b), a), x)' # f1 = input('f1 : ') # f2 = input('f2 : ')

result = unify(f1, f2) if not result:

print('The process of Unification failed!') else:

print('The process of Unification successful!') print(result)

## OUTPUT SCREENSHOTS:



# b) RESOLUTION ALGORITHM

**NAME: Rahul Goel**

**REG NO:** RA1911030010094

**AIM**: To implement resolution algorithm.

**PROCEDURE**:
Resolution is used, if there are various statements are given, and we need to prove a conclusion of those statements. Unification is a key concept in proofs by resolutions. Resolution is a single inference rule which can efficiently operate on the **conjunctive normal form or clausal form.**

- 1) Conversion of facts into first-order logic.
- 2) Convert FOL statements into CNF
- 3) Negate the statement which needs to prove (proof by contradiction)
- 4) Draw resolution graph (unification).

**INPUT 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)) 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:

except:

>1:

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)

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:
```

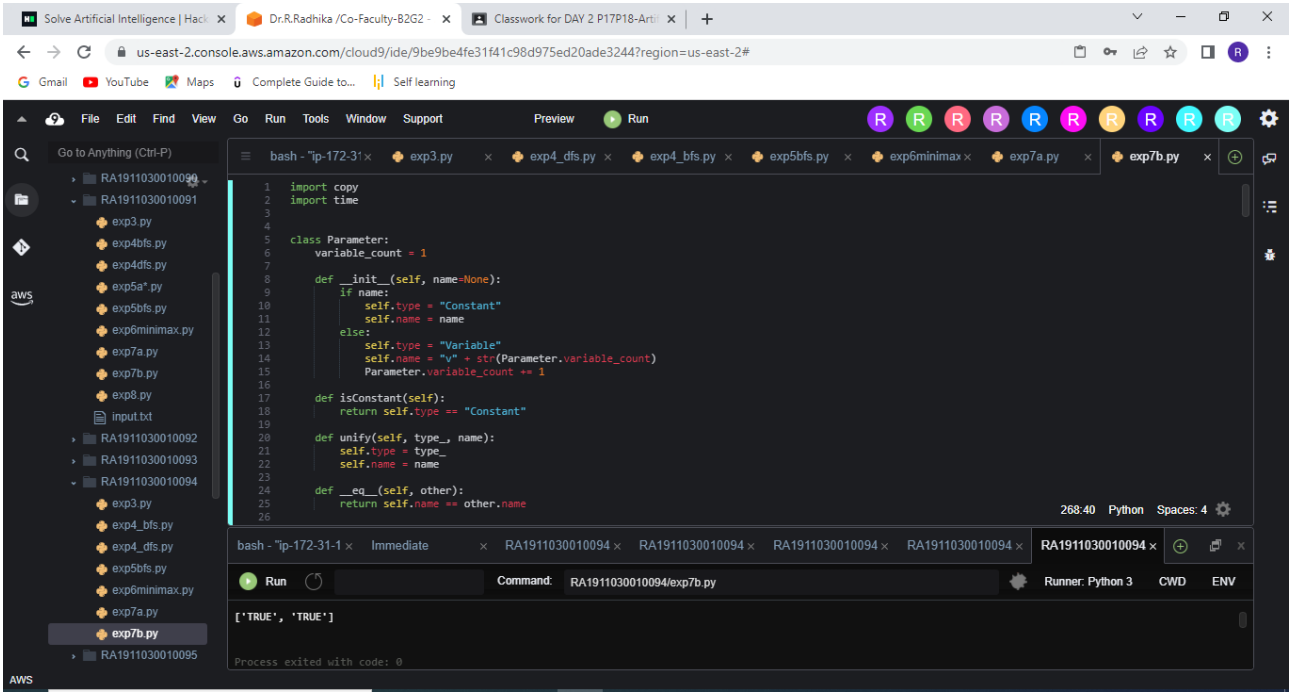
```
continue
if kb.isConstant():

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)
```

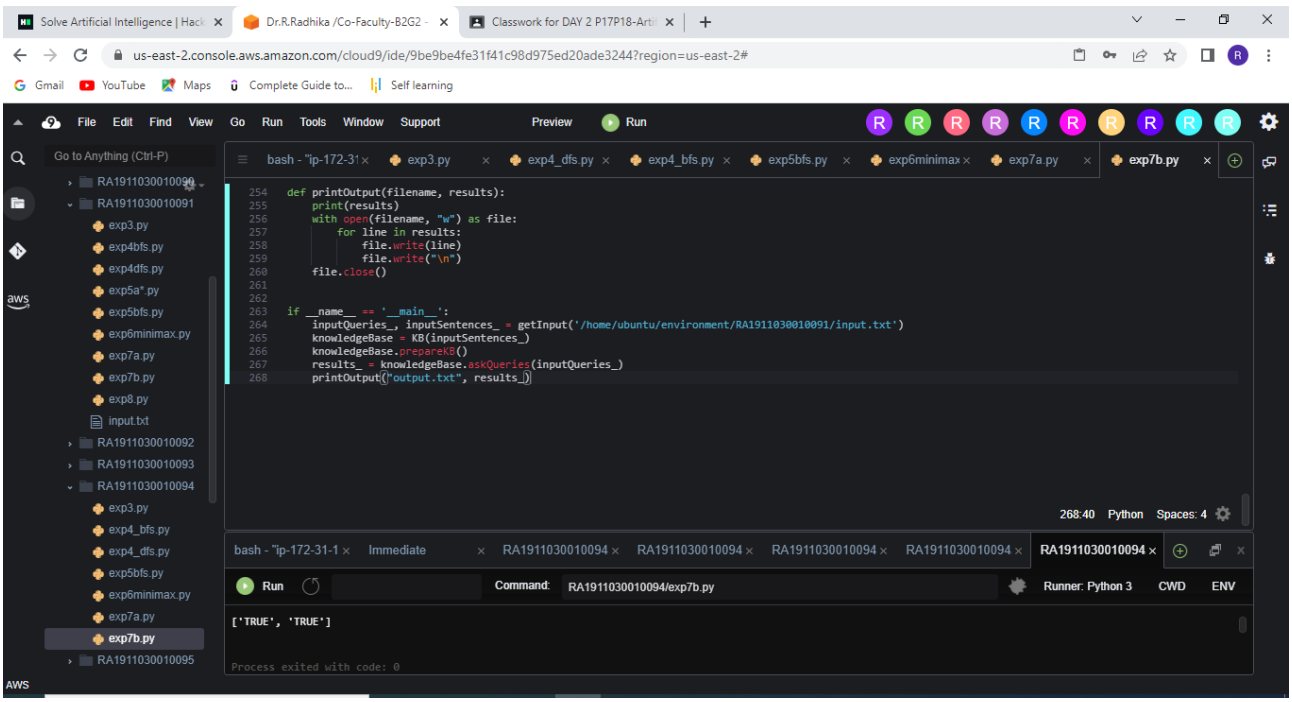
```
def getInput(filename):
with open(filename, "r") as file:

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()

name == ' main ': inputQueries_, inputSentences_ =

if

getInput('/home/ubuntu/environment/104/input.txt') knowledgeBase
= KB(inputSentences_) knowledgeBase.prepareKB()
results_ = knowledgeBase.askQueries(inputQueries_)
printOutput("output.txt", results_)
```

**INPUT.txt code:**

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

**OUTPUT SCREENSHOTS:**


Solve Artificial Intelligence | Hack
Dr.R.Radhika /Co-Faculty-B2G2 -
Classwork for DAY 2 P17P18-Artif
us-east-2.console.aws.amazon.com/cloud9/ide/9be9be4fe31f41c98d975ed20ade3244?region=us-east-2#
Gmail
YouTube
Maps
Complete Guide to...
Self learning
File Edit Find View Go Run Tools Window Support
Preview
Run
Go to Anything (Ctrl-P)
RA1911030010090
RA1911030010091
exp3.py
exp4bfs.py
exp4dfs.py
exp5a*.py
exp5bfs.py
exp6minimax.py
exp7a.py
exp7b.py
exp8.py
input.txt
RA1911030010092
RA1911030010093
RA1911030010094
exp3.py
exp4_bfs.py
exp4_dfs.py
exp5bfs.py
exp6minimax.py
exp7a.py
exp7b.py
RA1911030010095
bash - "ip-172-31
exp3.py
exp4_dfs.py
exp4_bfs.py
exp5bfs.py
exp6minimax
exp7a.py
exp7b.py
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
268:40 Python Spaces: 4
bash - "ip-172-31-1
Immediate
RA1911030010094
RA1911030010094
RA1911030010094
RA1911030010094
RA1911030010094
Run
Command: RA1911030010094/exp7b.py
Runner: Python 3
CWD
ENV
['TRUE', 'TRUE']
Process exited with code: 0
AWS


Solve Artificial Intelligence | Hack
Dr.R.Radhika /Co-Faculty-B2G2 -
Classwork for DAY 2 P17P18-Artif
us-east-2.console.aws.amazon.com/cloud9/ide/9be9be4fe31f41c98d975ed20ade3244?region=us-east-2#
Gmail
YouTube
Maps
Complete Guide to...
Self learning
File Edit Find View Go Run Tools Window Support
Preview
Run
Go to Anything (Ctrl-P)
RA1911030010090
RA1911030010091
exp3.py
exp4bfs.py
exp4dfs.py
exp5a*.py
exp5bfs.py
exp6minimax.py
exp7a.py
exp7b.py
exp8.py
input.txt
RA1911030010092
RA1911030010093
RA1911030010094
exp3.py
exp4_bfs.py
exp4_dfs.py
exp5bfs.py
exp6minimax.py
exp7a.py
exp7b.py
RA1911030010095
bash - "ip-172-31
exp3.py
exp4_dfs.py
exp4_bfs.py
exp5bfs.py
exp6minimax
exp7a.py
exp7b.py
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/RA1911030010091/input.txt')
knowledgeBase = KB(inputSentences_)
knowledgeBase.prepareKB()
results_ = knowledgeBase.askQueries(inputQueries_)
printOutput("output.txt", results_)
268:40 Python Spaces: 4
bash - "ip-172-31-1
Immediate
RA1911030010094
RA1911030010094
RA1911030010094
RA1911030010094
RA1911030010094
Run
Command: RA1911030010094/exp7b.py
Runner: Python 3
CWD
ENV
['TRUE', 'TRUE']
Process exited with code: 0
AWS