AMIT SRIVASTAV
RA1911003010633
ARTIFICIAL INTELLIGENCE LAB
EXPERIMENT NO: 7

IMPLEMENTATION OF UNIFICATION AND RESOLUTION IN REAL WORLD PROBLEMS

❖ *Unification (Pattern matching):*

Algorithm:

Step-1: Start

Step-2: Declare a Python dict mapping variable names to terms

Step-3: When either side is a variable, it calls `unify_variable.`

Step-4: Otherwise, if both sides are function applications, it ensures they apply the same function (otherwise there's no match) and then unifies their arguments one by one, carefully carrying the updated substitution throughout the process.

Step-5: If `v` is bound in the substitution, we try to unify its definition with `x` to guarantee consistency throughout the unification process (and vice versa when `x` is a variable).

Step-6: `occurs_check`, is to guarantee that we don't have self-referential variable bindings like `X=f(X)` that would lead to potentially infinite unifiers.

Step-7: Stop

Source code:

```
# Python 3.6
import lexer

class Term:
   pass

# In App, function names are always considered to be constants, not variables.
# This simplifies things and doesn't affect expressivity. We can always model
# variable functions by envisioning an apply(FUNCNAME, ... args ...).
class App(Term):
   def __init__(self, fname, args=()):
      self.fname = fname
      self.args = args

   def __str__(self):
      return '{0}({1})'.format(self.fname, ','.join(map(str, self.args)))

   def __eq__(self, other):
      return (type(self) == type(other) and
            self.fname == other.fname and
            all(self.args[i] == other.args[i] for i in range(len(self.args))))

   __repr__ = __str__


class Var(Term):
   def __init__(self, name):
      self.name = name

   def __str__(self):
      return self.name

   def __eq__(self, other):
      return type(self) == type(other) and self.name == other.name

   __repr__ = __str__


class Const(Term):
   def __init__(self, value):
      self.value = value

   def __str__(self):
      return self.value

   def __eq__(self, other):
      return type(self) == type(other) and self.value == other.value
```

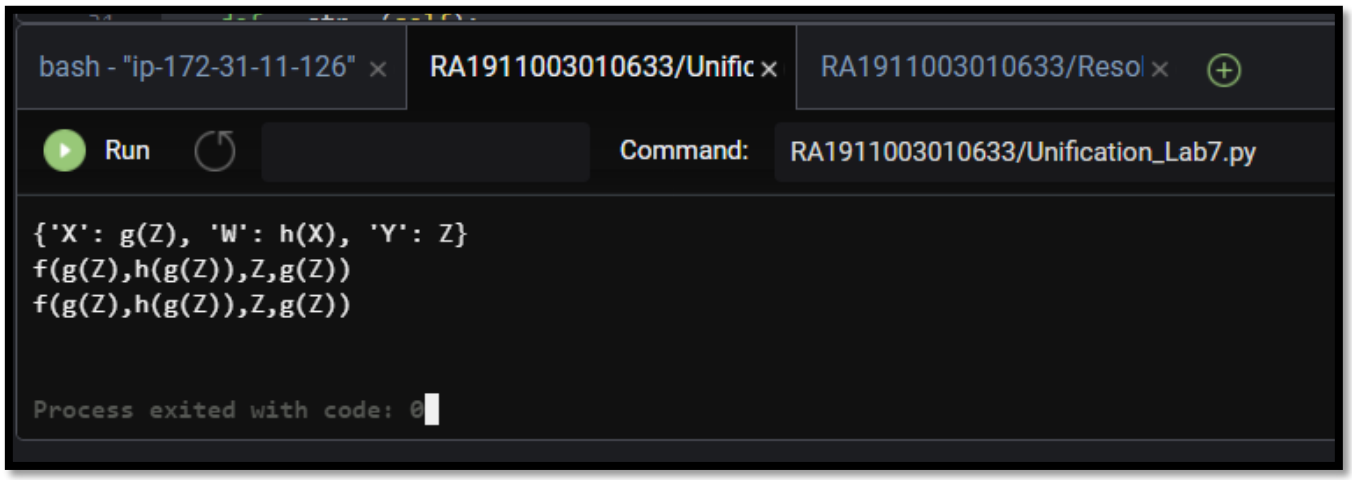
```
    __repr__ = __str__


class ParseError(Exception): pass

def parse_term(s):
    """Parses a term from string s, returns a Term."""
    parser = TermParser(s)
    return parser.parse_term()


class TermParser:
    """Term parser.

    Use the top-level parse_term() instead of instantiating this class directly.
    """
    def __init__(self, text):
        self.text = text
        self.cur_token = None
        lexrules = (
            ('\d+',           'NUMBER'),
            ('[a-zA-Z_]\w*',    'ID'),
            (',',             'COMMA'),
            ('\(',            'LP'),
            ('\)',            'RP'),
        )
        self.lexer = lexer.Lexer(lexrules, skip_whitespace=True)
        self.lexer.input(text)
        self._get_next_token()

    def _get_next_token(self):
        try:
            self.cur_token = self.lexer.token()

            if self.cur_token is None:
                self.cur_token = lexer.Token(None, None, None)
        except lexer.LexerError as e:
            self._error('Lexer error at position %d' % e.pos)

    def _error(self, msg):
        raise ParseError(msg)

    def parse_term(self):
        if self.cur_token.type == 'NUMBER':
            term = Const(self.cur_token.val)
            # Consume the current token and return the Const term.
            self._get_next_token()
            return term
        elif self.cur_token.type == 'ID':
```

```
            # We have to look at the next token to distinguish between App and
            # Var.
            idtok = self.cur_token
            self._get_next_token()
            if self.cur_token.type == 'LP':
                if idtok.val.isupper():
                    self._error("Function names should be constant")
                self._get_next_token()
                args = []
                while True:
                    args.append(self.parse_term())
                    if self.cur_token.type == 'RP':
                        break
                    elif self.cur_token.type == 'COMMA':
                        # Consume the comma and continue to the next arg
                        self._get_next_token()
                    else:
                        self._error("Expected ',' or ')' in application")
                # Consume the ')'
                self._get_next_token()
                return App(fname=idtok.val, args=args)
            else:
                if idtok.val.isupper():
                    return Var(idtok.val)
                else:
                    return Const(idtok.val)


def occurs_check(v, term, subst):
    """Does the variable v occur anywhere inside term?

    Variables in term are looked up in subst and the check is applied
    recursively.
    """
    assert isinstance(v, Var)
    if v == term:
        return True
    elif isinstance(term, Var) and term.name in subst:
        return occurs_check(v, subst[term.name], subst)
    elif isinstance(term, App):
        return any(occurs_check(v, arg, subst) for arg in term.args)
    else:
        return False


def unify(x, y, subst):
    """Unifies term x and y with initial subst.

    Returns a subst (map of name->term) that unifies x and y, or None if
    they can't be unified. Pass subst={} if no subst are initially
```

```
    known. Note that {} means valid (but empty) subst.
    """
    if subst is None:
        return None
    elif x == y:
        return subst
    elif isinstance(x, Var):
        return unify_variable(x, y, subst)
    elif isinstance(y, Var):
        return unify_variable(y, x, subst)
    elif isinstance(x, App) and isinstance(y, App):
        if x.fname != y.fname or len(x.args) != len(y.args):
            return None
        else:
            for i in range(len(x.args)):
                subst = unify(x.args[i], y.args[i], subst)
            return subst
    else:
        return None


def apply_unifier(x, subst):
    """Applies the unifier subst to term x.

    Returns a term where all occurrences of variables bound in subst
    were replaced (recursively); on failure returns None.
    """
    if subst is None:
        return None
    elif len(subst) == 0:
        return x
    elif isinstance(x, Const):
        return x
    elif isinstance(x, Var):
        if x.name in subst:
            return apply_unifier(subst[x.name], subst)
        else:
            return x
    elif isinstance(x, App):
        newargs = [apply_unifier(arg, subst) for arg in x.args]
        return App(x.fname, newargs)
    else:
        return None


def unify_variable(v, x, subst):
    """Unifies variable v with term x, using subst.

    Returns updated subst or None on failure.
    """
```

```
    assert isinstance(v, Var)
    if v.name in subst:
        return unify(subst[v.name], x, subst)
    elif isinstance(x, Var) and x.name in subst:
        return unify(v, subst[x.name], subst)
    elif occurs_check(v, x, subst):
        return None
    else:
        # v is not yet in subst and can't simplify x. Extend subst.
        return {**subst, v.name: x}


if __name__ == '__main__':
    s1 = 'f(X,h(X),Y,g(Y))'
    s2 = 'f(g(Z),W,Z,X)'
    subst = unify(parse_term(s1), parse_term(s2), {})
    print(subst)

    print(apply_unifier(parse_term(s1), subst))
    print(apply_unifier(parse_term(s2), subst))
```

Output:



Result:

Hence, the Implementation of unification algorithm for Pattern Matching is done successfully.

❖ *Resolution (Predicate logic):*

Algorithm:

Step-1: Start

Step-2: if L1 or L2 is an atom part of same thing do

(a) if L1 or L2 are identical then return NIL

(b) else if L1 is a variable then do

(i) if L1 occurs in L2 then return F else return (L2/L1)

else if L2 is a variable then do

(i) if L2 occurs in L1 then return F else return (L1/L2)

else return F.

Step-3: If length (L!) is not equal to length (L2) then return F.

Step-4: Set SUBST to NIL

(at the end of this procedure , SUBST will contain all the substitutions used to unify L1 and L2).

Step-5: For I = 1 to number of elements in L1 do

i) call UNIFY with the i th element of L1 and I'th element of L2, putting the result in S

ii) if S = F then return F

iii) if S is not equal to NIL then do

(A) apply S to the remainder of both L1 and L2

(B) SUBST := APPEND (S, SUBST) return SUBST.

Step-6: Stop.

Source 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)
```

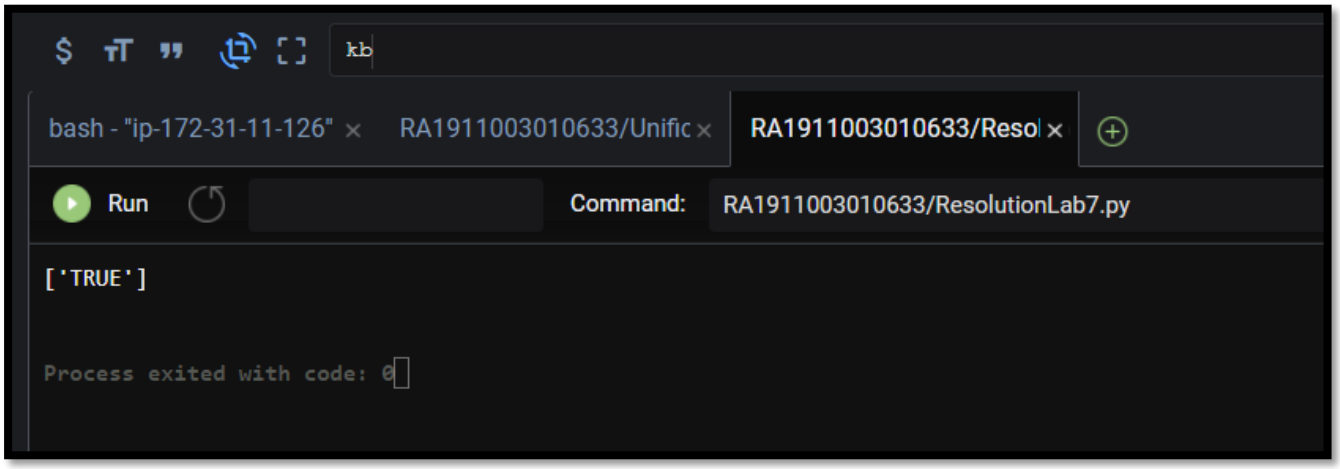
```
            self.sentence_map[negatedPredicate.name] = self.sentence_map.get(
                negatedPredicate.name, []) + [negatedQuery]
            self.timeLimit = time.time() + 40

            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('RA1911003010633/input1.txt')
    knowledgeBase = KB(inputSentences_)
    knowledgeBase.prepareKB()
    results_ = knowledgeBase.askQueries(inputQueries_)
    printOutput("output.txt", results_)
```

Output:



Result:

Hence, the Implementation of resolution algorithm for Predicate logic is done successfully.