Program 9: Convert given first order logic statement into Conjunctive Normal Form(CNF). Write_up:

-	PRANAV JAGADEESH IBM 18 CSO71 BATCH 2 5B PROGRAM 9
	Connect given find order logie into Conjuctive Normal Form (CNF):-
	import ne
	def remone-brackets (source, id):
	$aeg = ((([\land \land \land \land] *?))),$
	m = re. rearch (reg, course)
	of m in None:
+	return None, None
	new-source = re sule (reg, str (id), source, court = 1)
	neturn new source, m. group (1)
	class logue lease.
	def - init - (self, input):
	self. myestack = []
+	self. source = input
	fénel = input
+	while 1:
	infrit, trop = remove _ brackets (infrit, len (relf.my-stack))
	(relf.my-stack)
	if input in None.
-	break
1	final = what
	while 1.
-	input, timp = remove-brackets (input, lent.
	eelf. my_stack)
	if infrut 4 None:
	break
	final = input
	self. my_stack, append (tmp)
	self my stack append (final)

```
get-result (self)
def
      noot = self my stack [-1]

m = rematch ('\s * ([0-9]+)\s * $', soot)
      if m is not None
          root = self. my stack [int (m. group (1))]
      reg = ( (d+)
     mhile 2.
         m = re. warch (reg, root)
         if m is None:
          new = (('+ self. my_stack [ int (m.group(1))]+')
        Root = re. rub (reg, new, rest, sount =1)
 def merge tems (relf, logie)

reg 0 = '( \d + )'
           1 = ( neg \s+(\d+))
       for i in range (len (self. my stock))
target = self. my-etack [i]
           target = self, my-etack
          if logic not in target
                 ne rearch (reg 1, target
           m = ne. search (regl, target)
           if m ig None:
            for j in re. findall ( reg 0, target):

child = relf. my-strek [ int (j))
               if logic not in child:
continue

new-reg = "(1/5)"+j+"(15/4)"
```

self-my-stack [i] = ke sub (new neg, ''+thild+''

self-my-stack[i], count = 1) my-stack [i] = seeb my-stack[i]- strip() flag = Touce self. merge_items (logie) Mass ordering (logée base) def run (self): flag = False for i in range (len (self-my-stack) new_source = relf. add ibrackets (self.my-stack (i) if self. my_ stack[i] = now_ source self. my-stack[j] = new_source flag = Terue def add brashets (self, source) reg and = "(neg \S+) 2 \S+ \& + and \s+ (m. = ne search (reg and return re sub (reg and, "(" source, count = neg-or = " (neg \s +)? \s + \ n+ or \s f m. = re search (hag topy source return tre rule (reg - DH, "("
source, count = 1) reg_thip="(neg 1s+) 2 \s+ \n+ or \s+ (neg if m is not None.

```
retwin re sull(reg - somb, "("+ m. group (0) + ")",
source, count = 1)
        reg - iff = "(heg | st) ? | st | & t iff | & t (neg | st) 2 | st"
        m = re search (reg - iff, lowree)
             retroin re enle (reg-iff, "("+ m. group (0) +")",

Rowree, count = 1)
class replace if (logenbare)
          final = len ( reflace_all_iff() -)
flag = self. reflace_all_iff()
                . my stack - affend ( self. my stack (final ]
         replace all - iff (self)
              ans = self. replace_iff - inner ( self. my stack (i), I
                     my stack a spend (are (2)
      def replace - iff - inner (self, source, id):
            m = re search (reg , source
                 return None
            a, b = m. group (1), m. group (2)
return (str(id) + and '+ str(id+1)
```

tb, b+1 imp '+a) class replace _ mp (logie _ bace) def sun (self) blag - False i in range (len(self. my-stack)): ans - self. replace - imp_inner (self. my stock [i]) ef ans is None continue self-my-etack [i] = ans flag = True replace_imp_inner (self, source) reg = 1 1 (. + ? In timo) s+ (. + ?) \$1 = he reach (reg, source a, b = m. group (1), m. group (2) setuen a sreplace ('neg', sreturn 'neg' + a + 1 bs : + b class de-morgan (logéi_base): reg = 'neg \s + (\d+)' flag = False fénal = len (relf. mystark) -)
for i in range (len (relf. my-eback)): tanget = relf. my-stack [1]

m = re. rearch (reg, target) if m is None shild = self. Thy- stack [ant (m. group (1)))

```
self my stack [i] = ne sale (neg, str (len(self my stack)),
larget, count = 1)
    self. my-slack append (self. doing - de - morg an (child))
       b reak
   self my stack append (self my stack [final])
   return flag
def doing-de-morgan (self, source):
item = re-uplit ('95+', source
    new_items =[]
    for item in items :
      if item = = (ox '=
        new items append ('and')
      elif item = = (and!
          new - items == (100g):
      elif item= = (neg)
         new items, append ( 'neg')
      elif len (item. thip (1) > 0:
           new_items append ( 'neg')
           new_ictems append (item)
      for in range (len (new-items) -1):
              if new- items (i+1) = = (neg ':
                  new-items [i] = 11
                1 1 goën ([i for 1 in nev_items if len(i)
            meg = ( )d+)
            final = len ( relf. my-stack ) - 1
            for i in range (len (relf. my-stack)):

touget = relf. my-stack [i]
```

if or not in self. my-stack[i]: m = re search (reg, target) if m is None for j in re findall (reg, target soild = relf my-stack (int (j) ound not in child new-reg = [(1 \ 8) 1 + j+ "()& item = re split () stand \ s+1, child tmp-list = [itr (j) my-stack), len (self o my-stack) fler (ite. telf my-stack-affend (re-sub (new-+ item + 1, target). Ktief self. my stack[] = (and ! jain (tombolet) self . my_ stock- orphend (self. my- Hack (firal) class simplification (logie base old = self get-result (for i in range (len (self - my - stack)): # self. my stack [i] = self reducing final=self.my-etack[-1] self. my-stack [-1] = self-reducing - and (final neturn len (old)! = len (self get - surnet (

def reducing and (self, target):

if and not in target:

relum target

items = set (re. split ('\stand \ta', target))

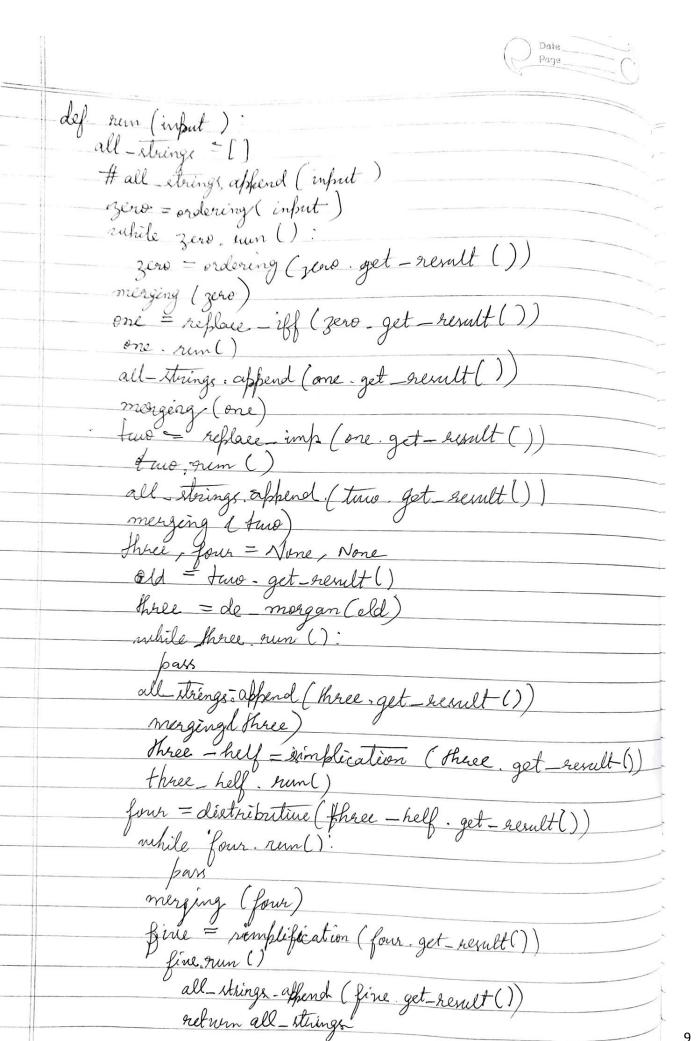
for item in list (items): if ('neg'+ item) în items: if he match (1)d + \$; item) is None: alue = self.my_Aack [int (item) if self.my_stack.count (value) value = 1 self my - Hack [int/item) return 'and' join (list (items)) if 'or not in larget of neturn larget of return larget of the split (' \ a + or \ a + ', target)) for item in list (items): if ('neg' + item) in iteams. netour or join (lit (items)) def merging (sourel):

old = source.get_nesult ()

source. Therge_items ('or')

source. merge_items ('and')

return cold (= source.get_result ()





enputs = input (). eplit ('\n')
for input in inputs
for item in our (input):
frint (item)
Enputs = input(). effect ('\n') for input in inputs for item in sum (input): fried (item) # out put. rusite ('\n')
Output:
-(Aiff c) or B
((A imp () and (C imp A)) or B ((neg A or C) and (neg C or A)) or B ((neg A or C) and (neg C or A)) or B ((neg C or A or B) and (neg A er B or C)
(neg A or C) and (neg C or A)) or B
((neg H or () and (neg (or A)) or B
(I neg Cos For B) and (neg A es B or C)

```
import re
def remove_brackets(source, id):
  reg = ' \setminus (([^ \setminus (]^*?) \setminus)'
  m = re.search(reg, source)
  if m is None:
     return None, None
  new_source = re.sub(reg, str(id), source, count=1)
  return new_source, m.group(1)
class logic_base:
  def __init__(self, input):
     self.my_stack = []
     self.source = input
     final = input
     while 1:
        input, tmp = remove_brackets(input, len(self.my_stack))
        if input is None:
          break
        final = input
        self.my_stack.append(tmp)
     self.my_stack.append(final)
  def get_result(self):
     root = self.my_stack[-1]
     m = re.match('\s^*([0-9]+)\s^*\$', root)
```

Program:

```
if m is not None:
     root = self.my_stack[int(m.group(1))]
  reg = '(\d+)'
  while 1:
     m = re.search(reg, root)
     if m is None:
       break
     new = '(' + self.my_stack[int(m.group(1))] + ')'
     root = re.sub(reg, new, root, count=1)
  return root
def merge_items(self, logic):
  reg0 = '(\d+)'
  reg1 = 'neg \s + (\d +)'
  flag = False
  for i in range(len(self.my_stack)):
     target = self.my_stack[i]
     if logic not in target:
       continue
     m = re.search(reg1, target)
     if m is not None:
       continue
     m = re.search(reg0, target)
     if m is None:
       continue
     for j in re.findall(reg0, target):
       child = self.my_stack[int(j)]
       if logic not in child:
          continue
```

```
new_reg = "(^{\s})" + j + "(\s|\$)"
          self.my_stack[i] = re.sub(new_reg, ' ' + child + ' ', self.my_stack[i], count=1)
          self.my_stack[i] = self.my_stack[i].strip()
          flag = True
     if flag:
       self.merge_items(logic)
class ordering(logic_base):
  def run(self):
     flag = False
    for i in range(len(self.my_stack)):
       new_source = self.add_brackets(self.my_stack[i])
       if self.my_stack[i] != new_source:
          self.my_stack[i] = new_source
          flag = True
     return flag
  def add brackets(self, source):
    reg = "\s+(and|or|imp|iff)\s+"
    if len(re.findall(reg, source)) < 2:
       return source
    reg_and = "(neg\s+)?\S+\s+and\s+(neg\s+)?\S+"
    m = re.search(reg_and, source)
     if m is not None:
       return re.sub(reg_and, "(" + m.group(0) + ")", source, count=1)
    reg\_or = "(neg\s+)?\S+\s+or\s+(neg\s+)?\S+"
     m = re.search(reg_or, source)
     if m is not None:
```

```
return re.sub(reg_or, "(" + m.group(0) + ")", source, count=1)
    reg_imp = "(neg\s+)?\S+\s+imp\s+(neg\s+)?\S+"
     m = re.search(reg imp, source)
     if m is not None:
       return re.sub(reg_imp, "(" + m.group(0) + ")", source, count=1)
    reg_iff = "(neg\s+)?\S+\s+iff\s+(neg\s+)?\S+"
     m = re.search(reg_iff, source)
     if m is not None:
       return re.sub(reg_iff, "(" + m.group(0) + ")", source, count=1)
class replace_iff(logic_base):
  def run(self):
    final = len(self.my_stack) - 1
     flag = self.replace_all_iff()
     self.my_stack.append(self.my_stack[final])
    return flag
  def replace_all_iff(self):
    flag = False
     for i in range(len(self.my_stack)):
       ans = self.replace_iff_inner(self.my_stack[i], len(self.my_stack))
       if ans is None:
          continue
       self.my_stack[i] = ans[0]
       self.my_stack.append(ans[1])
       self.my_stack.append(ans[2])
       flag = True
     return flag
```

```
def replace_iff_inner(self, source, id):
     reg = '^(.*?)\s+iff\s+(.*?)$'
     m = re.search(reg, source)
     if m is None:
       return None
     a, b = m.group(1), m.group(2)
     return (str(id) + 'and' + str(id + 1), a + 'imp' + b, b + 'imp' + a)
class replace_imp(logic_base):
  def run(self):
     flag = False
     for i in range(len(self.my_stack)):
       ans = self.replace_imp_inner(self.my_stack[i])
       if ans is None:
          continue
       self.my_stack[i] = ans
       flag = True
     return flag
  def replace_imp_inner(self, source):
     reg = '^(.*?)\s+imp\s+(.*?)$'
     m = re.search(reg, source)
     if m is None:
       return None
     a, b = m.group(1), m.group(2)
     if 'neg ' in a:
       return a.replace('neg', ") + ' or ' + b
```

```
return 'neg ' + a + ' or ' + b
```

```
class de_morgan(logic_base):
  def run(self):
     reg = 'neg \setminus s + (\setminus d +)'
     flag = False
     final = len(self.my_stack) - 1
     for i in range(len(self.my_stack)):
       target = self.my_stack[i]
       m = re.search(reg, target)
       if m is None:
          continue
       flag = True
       child = self.my_stack[int(m.group(1))]
       self.my_stack[i] = re.sub(reg, str(len(self.my_stack)), target, count=1)
       self.my_stack.append(self.doing_de_morgan(child))
       break
     self.my_stack.append(self.my_stack[final])
     return flag
  def doing_de_morgan(self, source):
     items = re.split('\s+', source)
     new_items = []
     for item in items:
       if item == 'or':
          new_items.append('and')
       elif item == 'and':
          new_items.append('or')
```

```
elif item == 'neg':
          new_items.append('neg')
       elif len(item.strip()) > 0:
          new_items.append('neg')
          new_items.append(item)
     for i in range(len(new_items) - 1):
       if new_items[i] == 'neg':
          if new_items[i + 1] == 'neg':
            new_items[i] = "
            new_items[i+1] = "
     return ''.join([i for i in new_items if len(i) > 0])
class distributive(logic_base):
  def run(self):
     flag = False
     reg = '(\d+)'
     final = len(self.my_stack) - 1
     for i in range(len(self.my_stack)):
       target = self.my_stack[i]
       if 'or' not in self.my_stack[i]:
          continue
       m = re.search(reg, target)
       if m is None:
          continue
       for j in re.findall(reg, target):
          child = self.my_stack[int(j)]
          if 'and' not in child:
             continue
```

```
new\_reg = "(^{\}\)" + j + "(\s|\$)"
          items = re.split('\s+and\s+', child)
          tmp list = [str(j) for j in range(len(self.my stack), len(self.my stack) +
len(items))]
          for item in items:
            self.my_stack.append(re.sub(new_reg, ' ' + item + ' ', target).strip())
          self.my_stack[i] = ' and '.join(tmp_list)
          flag = True
       if flag:
          break
     self.my_stack.append(self.my_stack[final])
     return flag
class simplification(logic_base):
  def run(self):
     old = self.get_result()
     for i in range(len(self.my_stack)):
       self.my_stack[i] = self.reducing_or(self.my_stack[i])
     # self.my_stack[i] = self.reducing_and(self.my_stack[i])
     final = self.my_stack[-1]
     self.my_stack[-1] = self.reducing_and(final)
     return len(old) != len(self.get_result())
  def reducing_and(self, target):
     if 'and' not in target:
       return target
     items = set(re.split('\s+and\s+', target))
     for item in list(items):
```

```
if ('neg ' + item) in items:
          return "
       if re.match('\d+$', item) is None:
          continue
       value = self.my_stack[int(item)]
       if self.my_stack.count(value) > 1:
          value = "
          self.my_stack[int(item)] = "
       if value == ":
          items.remove(item)
     return ' and '.join(list(items))
  def reducing_or(self, target):
     if 'or' not in target:
       return target
     items = set(re.split('\s+or\s+', target))
     for item in list(items):
       if ('neg ' + item) in items:
          return "
     return ' or '.join(list(items))
def merging(source):
  old = source.get_result()
  source.merge_items('or')
  source.merge_items('and')
  return old != source.get_result()
```

```
def run(input):
  all_strings = []
  # all_strings.append(input)
  zero = ordering(input)
  while zero.run():
     zero = ordering(zero.get_result())
  merging(zero)
  one = replace_iff(zero.get_result())
  one.run()
  all_strings.append(one.get_result())
  merging(one)
  two = replace_imp(one.get_result())
  two.run()
  all_strings.append(two.get_result())
  merging(two)
  three, four = None, None
  old = two.get_result()
  three = de_morgan(old)
  while three.run():
    pass
  all_strings.append(three.get_result())
  merging(three)
  three_helf = simplification(three.get_result())
  three_helf.run()
  four = distributive(three_helf.get_result())
```

```
while four.run():
   pass
 merging(four)
 five = simplification(four.get_result())
 five.run()
 all_strings.append(five.get_result())
 return all_strings
inputs = input().split('\n')
for input in inputs:
 for item in run(input):
   print(item)
 # output.write('\n')
Output:
(A iff C) or B
((A imp C) and (C imp A)) or B
((neg A or C) and (neg C or A)) or B
((neg A or C) and (neg C or A)) or B
(neg C or A or B) and (neg A or B or C)
Process finished with exit code 0
```