Program 7: Create a knowledgebase using prepositional logic and prove the given query using resolution.

Write\_up:

	PRANAV JAGADEESH IBMISCSDII PROGRAM 7
	Kb = []
	def CLEAR():
	global Kb Kb = []
	def TELL (sentence):
	gloleal Kb
	K6 = []
	def TELL (centence):
	gloleal the
	if to Clause (sentence).
	Kb. affend (slorlence)
	else:
	sentence(NF = convert CNF (sentence)
	if not sentence CNF:
	peint (" Ellegal Input!")
	nelwin
	if is And List (unteree CNF):
	for 5 in sentence (NF[1:]:
	Kb-append (s)
	elu:
	kb append (gentence (NF)
	def ASK (sentence):
	global Kb
	if is Clause ( Rentence)
	neg = negation (unterue)
	elle.
	gentence[NF = convert CNF (sentence)
	il not sentence CNF.
	phint ("Illegal input")
-	return
` ,	return  neg = connectCNF(negation (centence (NF))  ark_list = []
	ark_list = []



in intendical	
if isAndList(neg)	
for n in neg [1:]: $nCNF = makeCNF(n)$	
of type (ron CNF) hame == 'list':	
ask_list. invert(0, nlNF)	
che;	
ack_list; insert (o, a CNF)	
elre:	
ark_lit=[neg]	
Selanger = ackleit + Kb[:]	
gulile Thue:	
new_claus = [)	
for c1 in clauses:	
for the materies.	
if Cl is not in C2:	
nevolued = revolue (c1, c2)	
if resolved == False;	
continue	
if resolved == []:	
return Tome	
news dancer append (necolued)	
if ben new clauses ) == 0;	
netur Fale	
new_in_clauses = True	
for n in new-clauses:  if n not in clauser:	
if n not in clauser:	
new_in-clauses = False	
clauses, append (7)	
if new-in-clauses:	
retylin False	
The first of the	

def revolue (ary-one, ary-two); nevolued = False 1) = make - sentence (arg-one) 52 = make - sentence (arg-buro) desolue\_s[= None resolve \_ 52 = None for imyli if is Not List (i)= al= 1(1) af not: True al-not = False if a Not (int(j)): a 2 = j(1) a 2 - not = Thueelie: az \_not=False if a |-not | = a2 - not: resolved & Thue resolve 1 = i needler - 22 = j break if not revolued: return Falce

Classmate



 C rege.
elif not is clause ( unterex [i]):
 neturn False
 helwn Bake
return False
Adel negation (sentence):
Adel megation (sentence):
if is Clause (sentence):
 if is Clause (unterne): Preteurn Touce
elif is And List Crentence):
for 5 in sentence (1:):
if not is Clause (5):
return False
sutarn true
return False
def negation (sentence):
def negation (sentence):  if is Literal (sent ence):
return (not, sentence)
if is Not List (sentence);
return sentence []
if is AND[int (sentence):  neurlt = ('or')
result = ( or )
for i in rentence [I:]:
if is Not Liet (sentence):
result append (i-[])
elie:
result affent [ not', sentence])
return result
if in On List (sentence)
 result = ( and )
 for i in sentence [1:]
of in Nat Lin ( servence)
nesult. append (i(1))

```
che:
            result append ([not', electence ])
     return result
if is valist (centerce);
nexult = ['and']
   for i in sentence l']:
         if in NO Thist (centerice):
             result affend (i(1))
              herult append ([!not', i)
def convert CNF (centence):
     while not in CNF (sentence):
         rentence = make CNF (centence)
            if on f [0] == 'not if conf [1]

if on f [0] == 'not if conf [1]

if on f [0] == 'or';

return Typicke CNF (conf [1])

if on f [0] == 'or';

return to ['and']
                 for i in range (/, len (cn f)):
```



```
result append (make CNF ( t'not!, enf [i]]))
if cref [o] = = 'and'
result = ['or']
   for i in range (', len (cnf)):
result append (make CNF (['not', cnf [i]]))
    operand == 'implies' and len (sentence)
              make(NF(['or', ['not', make CN
                      (2))], make CNF (sentence
    frehand == 6\(\vec{\pi}\) cond\(\vec{\pi}\) for and len(sentence) == 3

&f = make CNF ([ 'implies', sentence [1], lentence [2]]

$\(\lambda\) = make (NF([ 'implies' s lentence [2], sentence [1]])
     neturn make CNF ( [ and ], sol
if is And List ( sentence
             in hange (1, len (sentence)

enf = make (NF (sentence [i])

il in Pudlist (cnl):
            if is Applicat (conf):

for i in range (1, len (conf)):

result. append (make CNF (conf [i]))
           neult append (make CIVF (enf))
     if is On List (sentence,
              for i in range (1, len sentence):

enf = make CNF (unterne [i])
                      our List (onf):
                       for i in range ((, len(enf)):
result[append (make CNF (cnf[i]))
```

```
continue
result 1. oppend (make CNF (cnf))
rubile Truce:
- Elsult 2 = [land']
        and-dame - More
        for h in result 1:
if is And List (h):
                  and-elaure = 2
      result I semone (and clause
        for 0 in range (1, len (and-clause.)):

temp={ ox 1, and slause [; ]]

for 0 in natult1[1:]:
         temp append (make (NF(D))

Fresult 2. experd (make CNF (temp))

result 1 = make CNF (result 2)
      reteum None
        if type (item) __ name__ == stat!:

#if len (item) == 2:

#if item (0) == (not!:

return True
     def is Not list (item):

if type (item) == ==!list':

if len(item) == 2:

if item(o) == (net':

hetron True
                       return False
```

	Date
	def is And Fut (item):
	if type (item)_name == 'list':
	if len (item) > 2:
\	if $item(0) = -(and)$ ,
`	seetwon False
	" / acc
	def is dr (it (item):
_	def is dr (it (item):  if type (item)name_ == 'list';  if len (item) > 2:
	if len (item) >2:
	if item[0] = = cor!
	refurn t-alpl
	O.H.AI = (IFARI)
	TELL (['implies', (p', 'ay'])  TELL (['implies', (p', 's'])  PSK ['implies', ['or', 'p', 's'], ('or', 'ay', 's']])
	THE ( mplies , ( n , (s ) )
	1 True
	<b>,</b> , , , , , , , , , , , , , , , , , ,
Ontput	CLEAR ()
,	TELL (l'implies', ['and', p', (ay'), 'n'])  TELL (l'implies', ['ar', (s', (t'), (ay')))
	TELL [ Implied   Car (c) (4)
	TELL ('t')
	ASK ('A')
	True
Output 3	CLEAR ()
	TELL ('b')
1	TELL ((c))
	7511 (41)
	ASR([(as',(a',(b',(c),(d'))

		Page
		X
Outfut 4	CLEAR()	
	CLEAR() TELL(a')	
	Tell (b')	
	TELL(((on), ['hot', (a)], (b)])	
	TELE ( ( on ' c', d')	
	Tell(b')  Tell(con', ['hot', (a'), (b')])  Tell(con', c', d')  Tell(d')  Ask ('c')	
	ASK ( cc	
	False	
		1

```
Program:
kb = []
def CLEAR():
  global kb
  kb = []
def TELL(sentence):
  global kb
  if isClause(sentence):
    kb.append(sentence)
  else:
    sentenceCNF = convertCNF(sentence)
    if not sentenceCNF:
       print("Illegal input")
       return
    if isAndList(sentenceCNF):
       for s in sentenceCNF[1:]:
         kb.append(s)
    else:
       kb.append(sentenceCNF)
def ASK(sentence):
  global kb
  if isClause(sentence):
    neg = negation(sentence)
```

```
else:
  sentenceCNF = convertCNF(sentence)
  if not sentenceCNF:
     print("Illegal input")
     return
  neg = convertCNF(negation(sentenceCNF))
ask_list = []
if isAndList(neg):
  for n in neg[1:]:
     nCNF = makeCNF(n)
    if type(nCNF).__name__ == 'list':
       ask_list.insert(0, nCNF)
     else:
       ask_list.insert(0, nCNF)
else:
  ask\_list = [neg]
clauses = ask_list + kb[:]
while True:
  new\_clauses = []
  for c1 in clauses:
     for c2 in clauses:
       if c1 is not c2:
          resolved = resolve(c1, c2)
          if resolved == False:
            continue
         if resolved == []:
            return True
```

## new\_clauses.append(resolved)

```
if len(new_clauses) == 0:
       return False
    new_in_clauses = True
    for n in new_clauses:
       if n not in clauses:
         new_in_clauses = False
         clauses.append(n)
    if new_in_clauses:
       return False
  return False
def resolve(arg_one, arg_two):
  resolved = False
  s1 = make_sentence(arg_one)
  s2 = make_sentence(arg_two)
  resolve\_s1 = None
  resolve\_s2 = None
  for i in s1:
    if isNotList(i):
       a1 = i[1]
       a1\_not = True
    else:
       a1 = i
```

```
a1\_not = False
  for j in s2:
     if isNotList(j):
       a2 = j[1]
       a2\_not = True
     else:
       a2 = j
       a2\_not = False
     if a1 == a2:
       if a1_not != a2_not:
          if resolved:
             return False
          else:
             resolved = True
            resolve\_s1 = i
            resolve\_s2 = j
             break
if not resolved:
  return False
s1.remove(resolve_s1)
s2.remove(resolve_s2)
result = clear\_duplicate(s1 + s2)
if len(result) == 1:
  return result[0]
elif len(result) > 1:
```

```
result.insert(0, 'or')
  return result
def make_sentence(arg):
  if isLiteral(arg) or isNotList(arg):
     return [arg]
  if isOrList(arg):
     return clear_duplicate(arg[1:])
  return
def clear_duplicate(arg):
  result = []
  for i in range(0, len(arg)):
     if arg[i] not in arg[i+1:]:
       result.append(arg[i])
  return result
def isClause(sentence):
  if isLiteral(sentence):
     return True
  if isNotList(sentence):
     if isLiteral(sentence[1]):
       return True
     else:
       return False
  if isOrList(sentence):
     for i in range(1, len(sentence)):
       if len(sentence[i]) > 2:
          return False
```

```
elif not isClause(sentence[i]):
          return False
     return True
  return False
def isCNF(sentence):
  if isClause(sentence):
     return True
  elif isAndList(sentence):
     for s in sentence[1:]:
       if not isClause(s):
          return False
     return True
  return False
def negation(sentence):
  if isLiteral(sentence):
     return ['not', sentence]
  if isNotList(sentence):
     return sentence[1]
  if isAndList(sentence):
     result = ['or']
     for i in sentence[1:]:
       if isNotList(sentence):
          result.append(i[1])
       else:
          result.append(['not', sentence])
     return result
  if isOrList(sentence):
```

```
result = ['and']
     for i in sentence[:]:
       if isNotList(sentence):
          result.append(i[1])
       else:
          result.append(['not', i])
     return result
  return None
def convertCNF(sentence):
  while not is CNF (sentence):
     if sentence is None:
       return None
     sentence = makeCNF(sentence)
  return sentence
def makeCNF(sentence):
  if isLiteral(sentence):
     return sentence
  if (type(sentence).__name__ == 'list'):
     operand = sentence[0]
     if isNotList(sentence):
       if isLiteral(sentence[1]):
          return sentence
       cnf = makeCNF(sentence[1])
       if cnf[0] == 'not':
          return makeCNF(cnf[1])
       if cnf[0] == 'or':
          result = ['and']
```

```
for i in range(1, len(cnf)):
       result.append(makeCNF(['not', cnf[i]]))
     return result
  if cnf[0] == 'and':
     result = ['or']
     for i in range(1, len(cnf)):
       result.append(makeCNF(['not', cnf[i]]))
     return result
  return "False: not"
if operand == 'implies' and len(sentence) == 3:
  return makeCNF(['or', ['not', makeCNF(sentence[1])], makeCNF(sentence[2])])
if operand == 'biconditional' and len(sentence) == 3:
  s1 = makeCNF(['implies', sentence[1], sentence[2]])
  s2 = makeCNF(['implies', sentence[2], sentence[1]])
  return makeCNF(['and', s1, s2])
if isAndList(sentence):
  result = ['and']
  for i in range(1, len(sentence)):
     cnf = makeCNF(sentence[i])
     if isAndList(cnf):
       for i in range(1, len(cnf)):
          result.append(makeCNF(cnf[i]))
       continue
     result.append(makeCNF(cnf))
  return result
```

```
if isOrList(sentence):
  result1 = ['or']
  for i in range(1, len(sentence)):
    cnf = makeCNF(sentence[i])
     if isOrList(cnf):
       for i in range(1, len(cnf)):
         result1.append(makeCNF(cnf[i]))
       continue
     result1.append(makeCNF(cnf))
  while True:
    result2 = ['and']
     and_clause = None
    for r in result1:
       if isAndList(r):
         and_clause = r
         break
    if not and_clause:
       return result1
     result1.remove(and_clause)
     for i in range(1, len(and_clause)):
       temp = ['or', and_clause[i]]
       for o in result1[1:]:
         temp.append(makeCNF(o))
       result2.append(makeCNF(temp))
     result1 = makeCNF(result2)
  return None
```

## return None

```
def isLiteral(item):
  if type(item).__name__ == 'str':
     return True
  return False
def isNotList(item):
  if type(item).__name__ == 'list':
     if len(item) == 2:
       if item[0] == 'not':
          return True
  return False
def isAndList(item):
  if type(item).__name__ == 'list':
     if len(item) > 2:
       if item[0] == 'and':
          return True
  return False
def isOrList(item):
  if type(item).__name__ == 'list':
     if len(item) > 2:
       if item[0] == 'or':
          return True
  return False
```

## Output:

```
CLEAR()
 #Test1
TELL(['implies', 'p', 'q'])
TELL(['implies', 'r', 's'])
ASK(['implies',['or','p','r'], ['or', 'q', 's']])
True
 CLEAR()
 #Test2
 TELL('p')
 TELL(['implies',['and','p','q'],'r'])
TELL(['implies',['or','s','t'],'q'])
 TELL('t')
 ASK('r')
True
CLEAR()
#Test3
TELL('a')
TELL('b')
TELL('c')
TELL('d')
ASK(['or', 'a', 'b', 'c', 'd'])
True
CLEAR()
#Test4
TELL('a')
TELL('b')
TELL(['or', ['not', 'a'], 'b'])
TELL(['or', 'c', 'd'])
TELL('d')
ASK('c')
False
```

alsc