```
import re

print("Enter FOL")
def remove_brackets(source, id):
    reg = '\(([^\(]*?)\)'
    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)
        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\s+(\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 = "(^|\s)" + 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:

```
==== RESTART: C:/Users/hp/AppData/Local/Programs/Python/Python310/FOL_CNF.py ===
Enter FOL
(animal(z) and kill (x,z)) imp (neg Loves(y,z))
(animal(z) and kill (x,z)) imp (neg Loves(y,z))
neg (animal(z) and kill (x,z)) or (neg Loves(y,z))
(neg animal(z) or neg kill (neg x,z)) or (neg Loves(y,z))
neg kill (neg x,z) or (neg Loves(y,z)) or neg animal(z)
>>>
```