Shiva Praneeth. Kodali
1BM18CS100

AI Lab Test - 2

28/12/2020 9:00am

Shivakodali

AI LAB TEST - 2

FOL => CNF

Code:

```
import re

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:
```

Shiva Praneeth. Kodeli
1BM18CS100

```
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))]
        rg = '(\d+)'
        while 1:
            m = re.search(rg, root)
            if m is None
                break
            new = '(' + self.my_stack[int(m.group(1))]
                  + ')'
            root = re.sub(rg, new, root, count=1)
        return root
```

```
def merge_items(self, logic):
    rego = '(\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(rego, target)
    if m is None:
        continue
    for j in re.findall(rego, 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)
    if flag:
```

```
self.mug_items(logic)

class ordering(logic_base)
    def run(self)
        flag = false
        for i in range(len(self.my_stack)):
            new_sourc = self.add_brackets(self.my_stack[i])
            if self.my_stack[i] != new_sourc:
                self.my_stack[i] = new_sourc
                flag = True
        return Flag

def add_brackets(self, sorca):
    reg = "\s*(and|or|imp|iff)\s*"
    if len(re.findall(reg, sourc)) < 2:
        return sourc
    reg_and = "(neg\s*)?\s*\s*and s*(neg\s*)?\s*"
    m = re.search(reg_and, sourc)
    if m is not None:
        return re.sub(reg_and, "("+m.group(0)+")",
                      sourc, count=1)
    reg_or = "(neg\s*)?\s*\s*or s*(neg\s*)?\s*"
    m = re.search(reg_or, sourc)
    if m is not None:
```

```
return re.sub(reg_or, "(" + m.group(0) + ")"
                                search, count=1)

reg_imp = "(neg|st)? \st \st and st (neg|st)? \st"

m = re.search(reg_imp, source)

if m is not None:

return re.sub(reg_imp, "(" + 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 xplaa_iff_inn(self, sour, id):
        reg = '^(.*?)\siff\s(.*?)$'
        m = re.search(reg, sour)
        if m is None
            return
        a, b = m.group(1), m.group(2)
        return (str(id)+' and '+str(id+1), a+' imp '+b, b+' imp '+a)

class xplaa_imp(logic_base):
    def run(self):
        flag = False
        for i in range(len(self.my_stack)):
            ans = self.xplaa_imp_inn(self.my_stac[i])
            if ans is None:
                continue
            self.my_stack[i] = ans
            flag = True
        return flag
```

```
def replace_imp_inner(self, source):
    reg = r'n(.*?)\st implist(.*?)\s'
    m = r.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

def doing_de_morgan(self, source):
    item = re.split('\st', source)
    new_items = []
    for item in items:
        if item == 'or':
            neg_item.append('and')
        elif item == 'and':
            new_item.appen('or')
        elif item == 'neg':
            new_item.append('neg')
        elif item Len(item.strip()) > 0:
            new_item.append('neg')
```

```
        new_items.append(items)
    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 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])
        final = self.my_stack[-1]
        return len(old) != len(self.get_result())
    def reducing_and(self, target):
        if 'and' not in target:
            return target
        items = set(re.split('\sand\s', target))
        for item in list(items):
            return ''
        if re.match('\d+$', item) is None
```

```
continue

value = self.my_stack[int(item)]

if self.my_stack.count(value) > 1:

    value == '':

        item.remove(item)

    return 'and'.join(list(items))

def reducing_or(self, target):

    if 'or' not in target:

        return target

    items = set(x.split('\s+or\s+', target))

        for ('neg' + item) in items:

            return '1'

        return 'or'.join(list(items))

def merge(source):

    old = source.get_result()

    source.merge_items('or')

    source.merge_items('and')

    return old != source.get_result()


def run(input):

    all_strings = []

    zero = ordering(input)

    while zero.run():

        zero = ordering(zero.get.next())

        skip merging(zero)
```

```
one = xplace_iff (zero.get_result())
    one.run()
    all_strings.append (one.get_result())
     merge (one)

two = xplace_iff ( zero. get_result())
      two.run()
  all_strings.append (two.get_result())
       merge(two)

three, four = None, None
      old = two.get_result()
       three = demorgan(old)
      while three.run()
          pass

all_strings.append (three.get_result())
      merging (three)

four = distributive (three_half.get_result())
       while four.run()
          pass

      merging (four)

      five = simplification (four.get_result())
            five.run()
          all_strings.append (five.get_result())
            return all_strings
```

```
if __name__ == '__main__':
    print ("Enter FOL")
    inputs = input().split('\n')
    print ("Steps :")
    for input in inputs:
        for item in run(input):
            print(item)
```