北京航空航天大學
BEIHANG UNIVERSITY

应用离散数学
实验报告
集合运算器、集合恒等式证明器

院（系）名称	自动化科学与电气工程学院
专业名称	自动化
学生学号	16711094
学生姓名	李翰韬

2018 年 12 月

目录

一、概述.....3
二、程序功能简述.....3
1、集合运算器.....3
2、幂集计算器.....5
3、集合恒等式证明器.....6
三、流程图（以集合恒等式证明器为例）.....7
四、主要算法介绍.....8
1、运算化简算法.....8
2、基于二进制的幂集算法.....10
3、集合恒等式证明器算法.....10
4、公式合法性检查代码.....12
五、 总结.....14

一、概述

本程序分为三个功能，分别为集合运算器、幂集计算器、集合恒等式证明器，对应了大作业文档中第一、二两项功能。下图为程序功能选择界面。

二、程序功能简述

1、集合运算器

上图集合运算器界面。

使用方法如下：

①．在第一个输入框中输入待插入集合中的元素。

②．点击下方五个按钮之一，将待插入元素插入某个集合中。插入后，集合中元素自动以逗号分隔。

③．集合元素设定完毕后（若未设定全集 E，则自动以 A∪B∪C∪D 代替，以免取反运算出现错误），在第二个输入框中使用下方按钮输入要计算的公式。

④．点击计算，若公式输入不符合规则，则程序报错，提醒用户进行更改；若公式符合规则，则程序运行结果自动出现在下方，元素用空格分隔。

以下为程序对经典算式的运行结果示例：

2、幂集计算器

上图为幂集计算器界面。

使用方法如下：

①．将待插入集合中的元素键入第一个输入框中。

②．点击下方按钮，将元素插入集合。

③．点击计算按钮，计算结果将显示在下方，元素用空格分隔，自动换行。

以下为程序对经典集合运行示例：

3、集合恒等式证明器

上图为集合恒等式证明器界面。

使用方法如下：

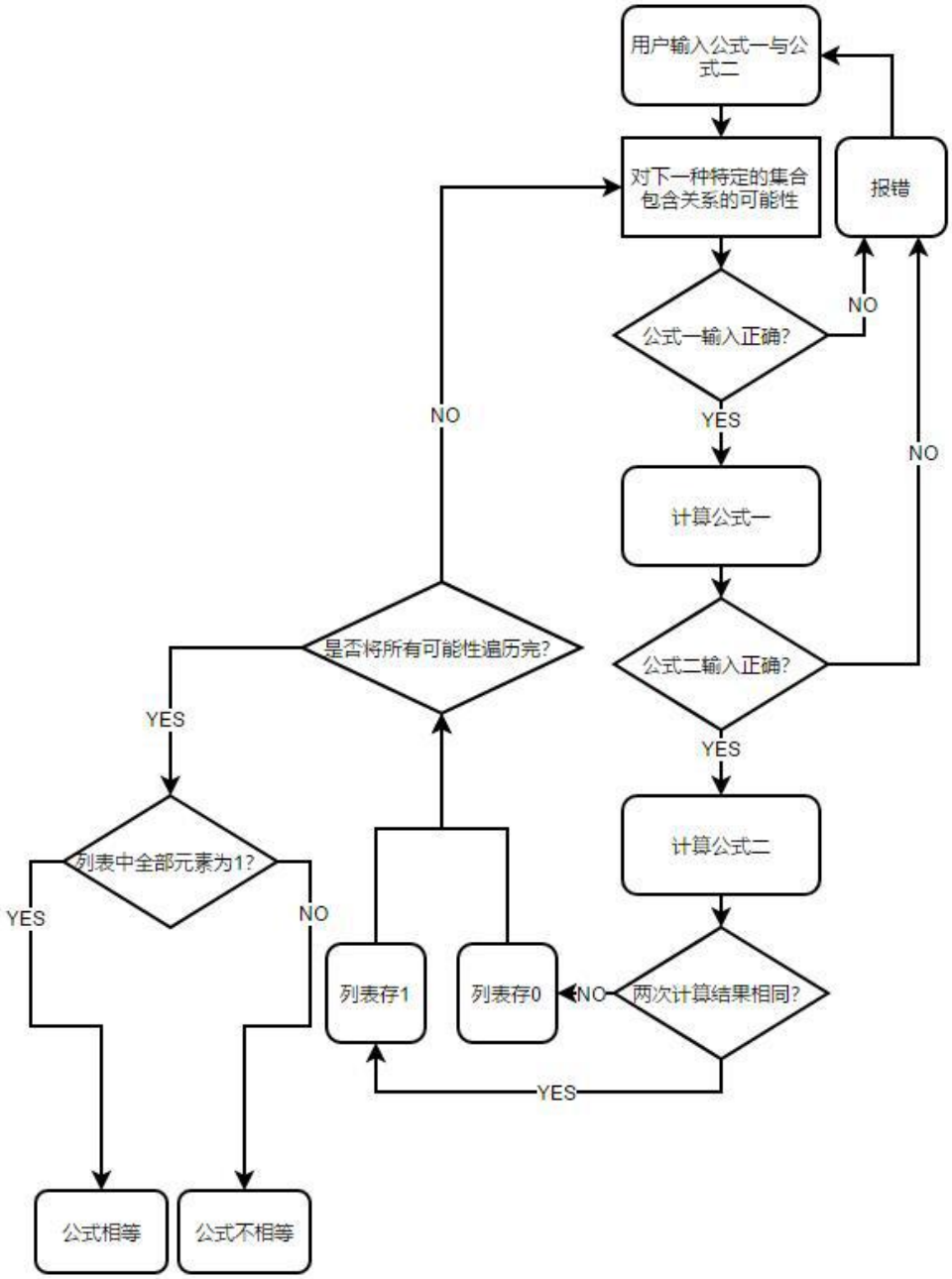
①. 将想验证相等的两个集合运算分别使用按钮键入公式一、公式二的输入框。

②. 点击验证，若公式输入不符合规则，则程序报错，提醒用户进行更改；若公式符合规则，则程序运行结果自动出现在下方。

以下为程序对经典恒等式的运行结果示例：

三、流程图（以集合恒等式证明器为例）

因为集合恒等式证明器的算法为前两项的结合（之后会解释），故只展示集合恒等式证明器的流程图。



四、主要算法介绍

1、运算化简算法

此算法为此程序的核心算法，作用为化简集合运算式，并判断优先级，最终得出集合运算式最终结果。算法大致分为以下几个步骤：

前提步骤：将需用到的集合提前设置好元素，定义运算。

以集合 A 为例：

```
strA = setA.get()
A0 = strA.split(',')
setA0 = set(A0)
A = list(setA0)
if len(A)==1:
    if A[0]=='∅':
        A = []
```

定义运算：

```
def calculate(set1,set2,operator,E):  #定义运算
    result = 0
    if operator == 'U':
        result = list(set(set1).union(set(set2)))
    if operator == 'n':
        result = list(set(set1).intersection(set(set2)))
    if operator == '⊕':
        result = list((set(set1).difference(set(set2))).
                      union(set(set2).difference(set(set1))))
    if operator == '-':
        result = list(set(set1).difference(set(set2)))
    if operator == '~':
        result = list(set(E).difference(set(set1)))
    return result
```

①.对输入的算式字符串进行简化，将字符串变为每个字符一个元素的列表。

```
def formula_format(formula):
    formula = re.sub(' ','',formula)
    formulalist0 = list(formula)
```

②. 判断公式是否合法，判断算法之后单独介绍。

③. 将列表中集合符号替换为集合元素。

```
formulalistA = [A if x == 'A' else x for x in formulalist0]
formulalistB = [B if x == 'B' else x for x in formulalistA]
formulalistC = [C if x == 'C' else x for x in formulalistB]
formulalistD = [D if x == 'D' else x for x in formulalistC]
formulalist = [E if x == 'E' else x for x in formulalistD]
```

③. 设定两个栈，集合栈和符号栈，依次判断列表中字符。若为集合，直接入集合栈；若为运算符号，调用函数决定操作：

```
def decision(tail_op,now_op):
    rate1 = ['∪','∩','⊕','-']
    rate2 = ['~']
    rate3 = ['(']
    rate4 = [')']
#-1入栈   0一起弹出   1出栈
    if tail_op in rate1:
        if now_op in rate2 or now_op in rate3:
            return -1
        else:
            return 1
    #栈顶为低优先级运算，新运算符号为~或（入栈，同为低优先级则出栈
    elif tail_op in rate2:
        if now_op in rate3:
            return -1
        else:
            return 1
    #栈顶为~，只有（入栈，其他出栈
    elif tail_op in rate3:
        if now_op in rate4:
            return 0
        else:
            return -1
    #栈顶为（，除）以外全部入栈
    else:
        return -1
    #栈顶为），全部入栈
```

以上函数依据符号栈顶与新符号比较，根据优先级不同确定操作。

根据函数返回值，确定下一步操作：

```
elif errorcode == 0:
    set_stack = []
    op_stack = []
    for e in formula_list:
        operator = check_operator(e)
        if not operator:     #是集合，则入集合栈
            set_stack.append(e)
        else:                #是运算符号
            while True:
                if len(op_stack) == 0:
                    op_stack.append(e)
                    break          #符号栈长度为0，直接入栈

                tag = decision(op_stack[-1],e)
                if tag == -1:      #入栈
                    op_stack.append(e)
                    break
                elif tag == 0:
                    op_stack.pop()
                    break
                elif tag == 1:
                    op = op_stack.pop()
                    if op == '~':
                        set1 = set_stack.pop()
                        set_stack.append(calculate(set1,set1,op,E))
                    else:
                        set2 = set_stack.pop()
                        set1 = set_stack.pop()
                        set_stack.append(calculate(set1,set2,op,E))
```

其中，-1 表示将符号入栈；0 表示( )的特殊情况，即左右括号相遇，将左右括号一起删除；1 表示将集合栈中的栈顶两个集合弹出（取反运算弹出一个），与符号栈顶运算符一同运算，将运算结果存入集合栈顶。

操作后继续循环，直至大循环处理结束。

④. 有可能存在最后两个栈内仍存在元素的情况，若有，则再进行一遍计算：

```
while len(op_stack) != 0:
    op = op_stack.pop()
    if op == '~':
        set1 = set_stack.pop()
        set_stack.append(calculate(set1,set1,op,E))
    else:
        set2 = set_stack.pop()
        set1 = set_stack.pop()
        set_stack.append(calculate(set1,set2,op,E))
```

⑤．将集合栈内集合输出，此集合即为运算结果。

2、基于二进制的幂集算法

此算法通过等效为二进制的方法，可便捷地求得某一集合的幂集。

算法大致思路如下：

对于集合元素个数为 N 的集合，其幂集必然包含 2N 个元素。若能够将 2N 个元素与 2N 个数字对应，则能够避免递归，大大简化算法。则可取 N 位二进制数字的所有可能，将某一可能中为 1 的位数提取，与集合中相应位置的元素相组合，便得到幂集中一个集合元素。

用包含三个元素的集合举例：

假设对集合{1，2，3}求幂集，集合中存在三个元素，则对 3 位二进制数字取所有可能：

000→∅ ；001→{3}；010→{2}；011→{2，3}；

100→{1}；101→{1，3}；110→{1，2}；111→{1，2，3}

由此可成功得到集合的所有幂集元素。

具体体现于代码时，寻找二进制数字中为 1 的位数，采用将二进制数右移，若右移 j 位后数字除 2 后余 1，则原二进制数中第 j 位为 1（最低位为 0 位）

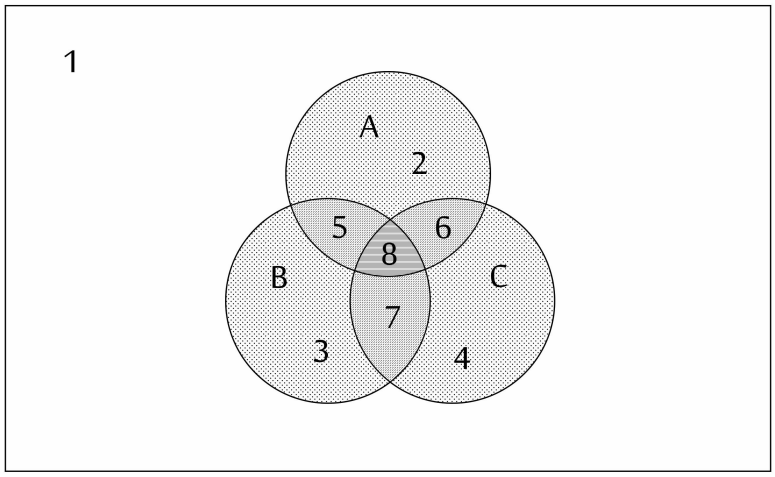
```
436         def PowerSetsBinary(items):
437             N = len(items)
438             set_all=[]
439 
440             for i in range(2**N):
441                 combo = []
442                 for j in range(N):
443                     if(i >> j) % 2 == 1:  #找到二级制数中为1的位数
444                         combo.append(items[j])
445                 set_all.append(combo)
446             return set_all
447 
448         out= PowerSetsBinary(A)
449         out[0] = 'ø'
450         var_answer.set(out)
```

上图为求幂集时的部分函数。

3、集合恒等式证明器算法

对于证明包含三个集合的集合算式相等，本程序借鉴文氏图，采用高效的遍历算法。

下图为三个集合的文氏图，对文氏图每个可能存在的区域标号 1~8。



由集合的基本定义可知，对于共同出现的三个集合，其状态可能性为上述不同区域的存在排列组合。

由此想到，可以用幂集计算器中求幂集的算法求得所有可能性。

下图为求所有集合包含关系可能性的算法语句以及运行结果，其中，若是某个∩区域存在元素，则两侧区域都得到元素。

```
for i in range(len(out)):
    A=[]
    B=[]
    C=[]
    AB=[]
    AC=[]
    BC=[]
    ABC=[]
    E=[]
    if ('1' in out[i]):
        E.append('E')
    if ('2' in out[i]):
        A.append('A')
        E.append('A')
    if ('3' in out[i]):
        B.append('B')
        E.append('B')
    if ('4' in out[i]):
        C.append('C')
        E.append('C')
    if ('5' in out[i]):
        A.append('AB')
        B.append('AB')
        AB.append('AB')
        E.append('AB')
    if ('6' in out[i]):
        A.append('AC')
        C.append('AC')
        AC.append('AC')
        E.append('AC')
    if ('7' in out[i]):
        B.append('BC')
        C.append('BC')
        BC.append('BC')
        E.append('BC')
    if ('8' in out[i]):
        A.append('ABC')
        B.append('ABC')
        C.append('ABC')
        AB.append('ABC')
        AC.append('ABC')
        BC.append('ABC')
        E.append('ABC')

    print('out=',out[i],'E=',list(set(E)),'A=',list(set(A)),'B=',list(set(B)),
          'C=',list(set(C)),'AB=',list(set(AB)),'AC=',list(set(AC)),'BC=',
          list(set(BC)),'ABC=',list(set(ABC)))
```

```
In [12]: runfile('/Users/hantao.li/Desktop/test.py', wdir='/Users/hantao.li/Desktop')
out= [] E= [] A= [] B= [] C= [] AB= [] AC= [] BC= [] ABC= []
out= ['1'] E= ['E'] A= [] B= [] C= [] AB= [] AC= [] BC= [] ABC= []
out= ['2'] E= ['A'] A= ['A'] B= [] C= [] AB= [] AC= [] BC= [] ABC= []
out= ['1', '2'] E= ['A', 'E'] A= ['A'] B= [] C= [] AB= [] AC= [] BC= [] ABC= []
out= ['3'] E= ['B'] A= [] B= ['B'] C= [] AB= [] AC= [] BC= [] ABC= []
out= ['1', '3'] E= ['E', 'B'] A= [] B= ['B'] C= [] AB= [] AC= [] BC= [] ABC= []
out= ['2', '3'] E= ['A', 'B'] A= ['A'] B= ['B'] C= [] AB= [] AC= [] BC= [] ABC= []
out= ['1', '2', '3'] E= ['A', 'E', 'B'] A= ['A'] B= ['B'] C= [] AB= [] AC= [] BC= [] ABC= []
out= ['4'] E= ['C'] A= [] B= [] C= ['C'] AB= [] AC= [] BC= [] ABC= []
out= ['1', '4'] E= ['E', 'C'] A= [] B= [] C= ['C'] AB= [] AC= [] BC= [] ABC= []
out= ['2', '4'] E= ['A', 'C'] A= ['A'] B= [] C= ['C'] AB= [] AC= [] BC= [] ABC= []
out= ['1', '2', '4'] E= ['A', 'E', 'C'] A= ['A'] B= [] C= ['C'] AB= [] AC= [] BC= [] ABC= []
out= ['3', '4'] E= ['B', 'C'] A= [] B= ['B'] C= ['C'] AB= [] AC= [] BC= [] ABC= []
out= ['1', '3', '4'] E= ['E', 'B', 'C'] A= [] B= ['B'] C= ['C'] AB= [] AC= [] BC= [] ABC= []
out= ['2', '3', '4'] E= ['A', 'B', 'C'] A= ['A'] B= ['B'] C= ['C'] AB= [] AC= [] BC= [] ABC= []
out= ['1', '2', '3', '4'] E= ['A', 'E', 'B', 'C'] A= ['A'] B= ['B'] C= ['C'] AB= [] AC= [] BC= [] ABC= []
out= ['5'] E= ['AB'] A= ['AB'] B= ['AB'] C= [] AB= ['AB'] AC= [] BC= [] ABC= []
out= ['1', '5'] E= ['AB', 'E'] A= ['AB'] B= ['AB'] C= [] AB= ['AB'] AC= [] BC= [] ABC= []
out= ['2', '5'] E= ['A', 'AB'] A= ['A', 'AB'] B= ['AB'] C= [] AB= ['AB'] AC= [] BC= [] ABC= []
out= ['1', '2', '5'] E= ['A', 'E', 'AB'] A= ['A', 'AB'] B= ['AB'] C= [] AB= ['AB'] AC= [] BC= [] ABC= []
out= ['3', '5'] E= ['AB', 'B'] A= ['AB'] B= ['AB', 'B'] C= [] AB= ['AB'] AC= [] BC= [] ABC= []
out= ['1', '3', '5'] E= ['AB', 'E', 'B'] A= ['AB'] B= ['AB', 'B'] C= [] AB= ['AB'] AC= [] BC= [] ABC= []
out= ['2', '3', '5'] E= ['A', 'B', 'AB'] A= ['A', 'AB'] B= ['AB', 'B'] C= [] AB= ['AB'] AC= [] BC= [] ABC= []
out= ['1', '2', '3', '5'] E= ['A', 'E', 'B', 'AB'] A= ['A', 'AB'] B= ['AB', 'B'] C= [] AB= ['AB'] AC= [] BC= [] ABC= []
out= ['4', '5'] E= ['AB', 'C'] A= ['AB'] B= ['AB'] C= ['C'] AB= ['AB'] AC= [] BC= [] ABC= []
out= ['1', '4', '5'] E= ['AB', 'E', 'C'] A= ['AB'] B= ['AB'] C= ['C'] AB= ['AB'] AC= [] BC= [] ABC= []
out= ['2', '4', '5'] E= ['A', 'C', 'AB'] A= ['A', 'AB'] B= ['AB'] C= ['C'] AB= ['AB'] AC= [] BC= [] ABC= []
out= ['1', '2', '4', '5'] E= ['A', 'E', 'C', 'AB'] A= ['A', 'AB'] B= ['AB'] C= ['C'] AB= ['AB'] AC= [] BC= [] ABC= []
out= ['3', '4', '5'] E= ['AB', 'B', 'C'] A= ['AB'] B= ['AB', 'B'] C= ['C'] AB= ['AB'] AC= [] BC= [] ABC= []
out= ['1', '3', '4', '5'] E= ['AB', 'E', 'B', 'C'] A= ['AB'] B= ['AB', 'B'] C= ['C'] AB= ['AB'] AC= [] BC= [] ABC= []
out= ['2', '3', '4', '5'] E= ['A', 'B', 'C', 'AB'] A= ['A', 'AB'] B= ['AB', 'B'] C= ['C'] AB= ['AB'] AC= [] BC= [] ABC= []
out= ['1', '2', '3', '4', '5'] E= ['A', 'AB', 'B', 'E', 'C'] A= ['A', 'AB'] B= ['AB', 'B'] C= ['C'] AB= ['AB'] AC= [] BC= [] ABC= []
out= ['6'] E= ['AC'] A= ['AC'] B= [] C= ['AC'] AB= [] AC= ['AC'] BC= [] ABC= []
out= ['1', '6'] E= ['AC', 'E'] A= ['AC'] B= [] C= ['AC'] AB= [] AC= ['AC'] BC= [] ABC= []
out= ['2', '6'] E= ['AC', 'A'] A= ['AC', 'A'] B= [] C= ['AC'] AB= [] AC= ['AC'] BC= [] ABC= []
out= ['1', '2', '6'] E= ['AC', 'A', 'E'] A= ['AC', 'A'] B= [] C= ['AC'] AB= [] AC= ['AC'] BC= [] ABC= []
out= ['3', '6'] E= ['AC', 'B'] A= ['AC'] B= ['B'] C= ['AC'] AB= [] AC= ['AC'] BC= [] ABC= []
```

其中，[ ]即为空集，即标号区域不存在元素。由上图可知，使用此种算法，很简便得得到了三个集合元素情况的所有可能性。

之后，每种集合情况下，对等式两边分别集合运算计算，若计算结果相同，则在结果列表中存 1，否则存 0。

将所有情况运行完毕后检查结果列表，若其中所有元素都为 1，则两个运算式相等，若其中任何元素为 0，则两个运算式不等。

```
919                 if set(result1[0]) == set(result2[0]):
920                     check.append(1)
921                 else:
922                     check.append(0)
923
924         if 0 in check:
925             answer.set('等式：“公式一 = 公式二” 不成立')
926         elif check == []:
927             answer.set('公式有误,请检查后重新输入!')
928         else:
929             answer.set('等式：“公式一 = 公式二” 成立')
```

上图为检查结果列表部分代码。

此算法在按照文氏图设定初始集合后，理论上可计算大于等于四个集合的算式证明，并不局限于三个集合。

4、公式合法性检查代码

此段代码意在对用户输入公式的合法性进行检查。若缺少此段代码，在用户键入正确公式时，不对整体程序产生任何影响；但若用户键入各种各样的错误公式，则会使程序产生不可预知的错误。

```
def GoCalculate():     #点击计算按钮
    True_list = ['A','B','C','D','E','∪','∩','⊕','-','~','(',')']
    Vital_list = ['∪','∩','⊕','-']
```

以上为下述代码中提到的两个列表的定义。

```
formulalist0 = list(formula)
if (set(formulalist0).issubset(set(True_list))):
    if (formula_list.count('(')) != (formula_list.count(')')):
        errorcode = 1
    else:
        while i < len(formula_list)-1:
            if (formula_list[i] != [] and set(formula_list[i]).issubset(set(Vital_list))):
                if i == 0:
                    errorcode = 4
                    i += 1
                    continue
                elif ((formula_list[i-1] == '(') or (formula_list[i+1] == ')')
        or (formula_list[i+1] != [] and set(formula_list[i+1]).issubset(set(Vital_list)))
        or (formula_list[i-1] != [] and set(formula_list[i-1]).issubset(set(Vital_list)))):
                    errorcode = 4
                    i += 1
                    continue
                else:
                    i += 1
                    continue
            elif formula_list[i] == '~':
                if formula_list[i+1] == ')' or formula_list[i-1] == ')':
                    errorcode = 4
                    i += 1
                    continue
                else:
                    i += 1
                    continue
            elif formula_list[i] == formula_list[i+1]:
                if formula_list[i] == '~' :
                    del formula_list[i]
                    del formula_list[i]
                elif ((formula_list[i] != '(') and (formula_list[i] != ')')):
                    errorcode = 2
                    i += 1
                    continue
                else:
                    i += 1
                    continue
            else:
                    i += 1
else:
    errorcode = 3
```

```
if errorcode == 1:
    err1.set('ERROR!公式一左右括号数量不等!请检查公式!')
elif errorcode == 2:
    err1.set('ERROR!公式一存在重复字符!请检查公式!')
elif errorcode == 3:
    err1.set('ERROR!公式一存在非法字符!请不要使用键盘输入!')
elif errorcode == 4:
    err1.set('ERROR!公式一不合法!请检查公式!')
#def final_clac(formula_list,A,B,C,D,E):
```

errorcode 为 1、2、3 时情况从错误显示报告中就可看出。对于 errorcode 为 4 的情况，分为三类：

①．运算符位于运算式第一个字符。

如：∩A∪B

②．两个运算符相连，或是运算符前有前括号，后有后括号。

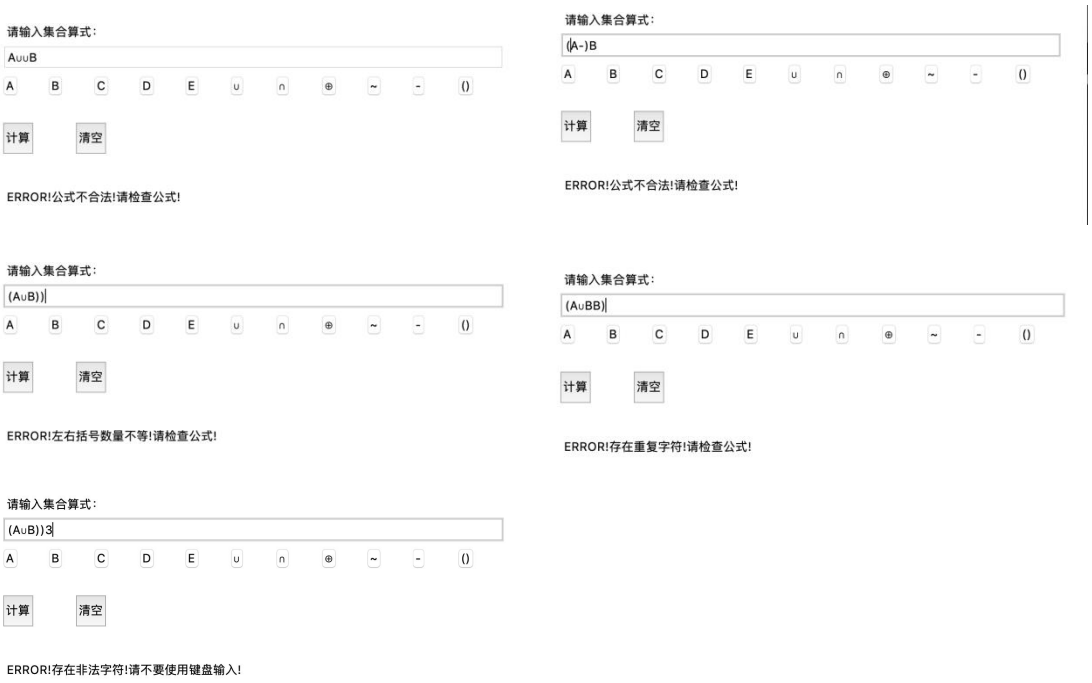
如：A∩∪B、（A∩）B、A（∪B）

③．~运算前后出现后括号。

如：（A∪B）~B、（A∪B~）∪B

我认为上述错误代码可涵盖大部分输入公式时出现的错误。为了减少错误的发生，本程序还将括号的输入直接定义为同时输入左括号与右括号，防止括号多层嵌套时用户不能准确分辨需要插入多少个右括号结尾。

代码判断示例如下：



五、总结

由于本人没有过多的系统化编程经验，故对于函数命名、变量命名等规则并不熟悉；且编程经历少，程序语句较为混乱，变量可能随用随定义，可能会造成可读性降低。希望能够参考文档阅读代码，从而便于代码理解，不胜感激！