**《离散数学》课程实验报告**

# 《离散数学》课程实验报告1

命题逻辑联接词、真值表、主范式

作 者 姓 名： 毛凌骏

学 号： 2053058

指 导 教 师： 唐剑锋

学院、 专业： 软件学院 软件工程

同济大学

Tongji University

1. **题目简介**

**1.1实验内容：**

A：从键盘输入两个命题变元P和Q的真值，求它们的合取、析取、条件和双向条件的真值；

B：求任意一个命题公式的真值表；

C：根据B所求真值表求主范式。

**1.2实验环境：**

采用C＋＋编程语言，Visual Studio Code 2019实验环境实现。

**1.3详细说明：**

**A题：**本实验要求利用C/C＋＋语言，实现二元合取、析取、条件和双向条件表达式的计算。充分利用联接词和逻辑运算符之间的相似性来实现程序功能。

**B、C题：**本实验要求利用C/C＋＋语言，实现任意输入公式的真值表计算,并通过查表求得命题公式的主析取范式与主合取范式。一般将公式中的命题变元放在真值表的左边，将公式的结果放在真值表的右边。命题变元可用数值变量表示，合式公式的表示及求真值表转化为逻辑运算结果；可用一维数表示合式公式中所出现的n个命题变元，同时它也是一个二进制加法器的模拟器，每当在这个模拟器中产生一个二进制数时，就相当于给各个命题变元产生了一组真值指派。算法逻辑如下：

（1）将二进制加法模拟器赋初值0。

（2）计算模拟器中所对应的一组真值指派下合式公式的真值。

（3）输出真值表中对应于模拟器所给出的一组真值指派及这组真值指派所对应的一行真值。

（4）产生下一个二进制数值，若该数值等于2n-1，则结束，否则转（2）。

1. **解题思路**

**2.1 A题：**

C语言中本身支持有与、或、非三种逻辑运算符，可以用&&、||、!来表示。对于两个命题变元P和Q，它们的合取式，析取式，因此直接调用C语言自带的逻辑运算功能即可实现。而命题变元P和Q的条件即蕴含式的真值以及P和Q的双向条件即等值式的真值无法直接得到，但根据离散数学命题逻辑知识，可知，,因此我们整理可得，,这样就将四个逻辑表达式全部转换成了C语言可处理的形式。

对于非布尔型数据，使用逻辑运算可能会出现问题，但我们可以使用位运算代替逻辑运算。位运算包括按位与&、按位或|、按位取反等~，正好对应逻辑运算中的&&、||、!。

[按位与](https://so.csdn.net/so/search?q=%E6%8C%89%E4%BD%8D%E4%B8%8E&spm=1001.2101.3001.7020" \t "https://blog.csdn.net/m0_58229878/article/details/_blank)是按二进制位与，比较两个二进制位数，如果上下同为0则为0，如果一个为1，一个为0则为0，如果同时为1则为1。故当非布尔型数值取0或1时，按位与&的运算结果与逻辑运算&的结果是一致的。

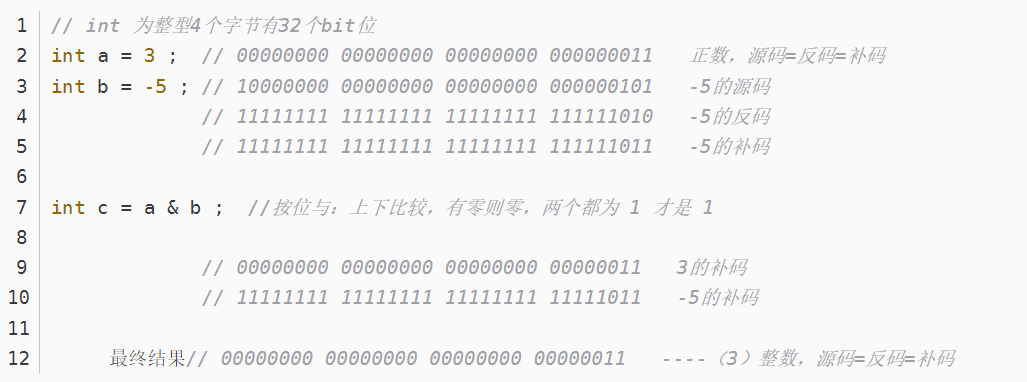


图 1 按位与示例

[按位或](https://so.csdn.net/so/search?q=%E6%8C%89%E4%BD%8D%E6%88%96&spm=1001.2101.3001.7020" \t "https://blog.csdn.net/m0_58229878/article/details/_blank)是按照二进制位或，比较两个二进制数，如果上下两个同为0，则为0，如果一个为1，一个为0，则结果为1，如果都是1则为1。当非布尔型数值取0或1时，按位或|的运算结果与逻辑运算||的结果是一致的。

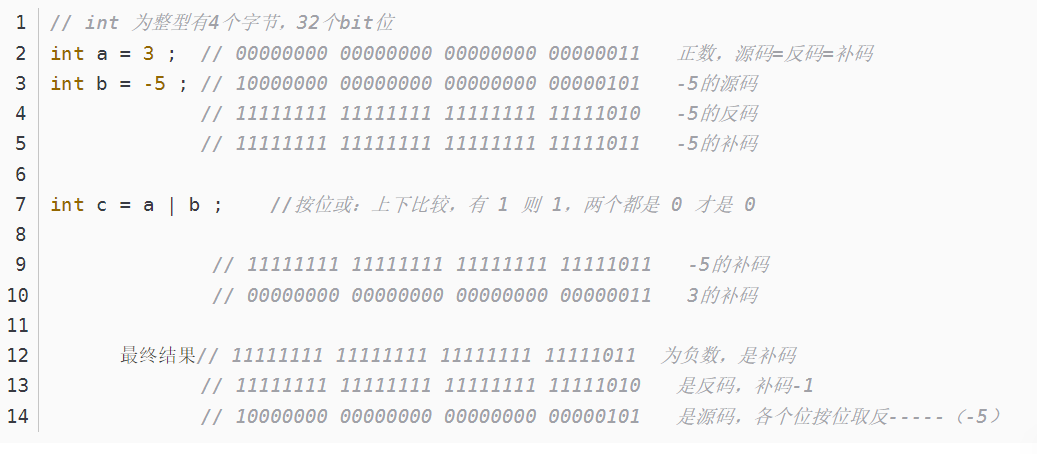


图 2 按位或示例

按位取反是位运算中的一元运算符，其主要功能是将二进制数的每一位进行取反，如果原先为0则结果为1，如果原先为1则结果为0，故当非布尔型数值取0或1时，按位取反~的运算结果与逻辑运算！的结果是一致的。

综上可得，实现逻辑运算可以通过逻辑运算符与位运算符两种方式实现，本文选择第一种方式。

**2.2 B题：**

真值表是表征逻辑事件输入和输出之间全部可能状态的表格。列出命题公式真假值的表。通常以1表示真，0 表示假。命题公式的取值由组成命题公式的命题变元的取值和命题联结词决定，命题联结词的真值表给出了真假值的算法。 真值表是在逻辑中使用的一类数学表，用来确定一个表达式是否为真或有效。

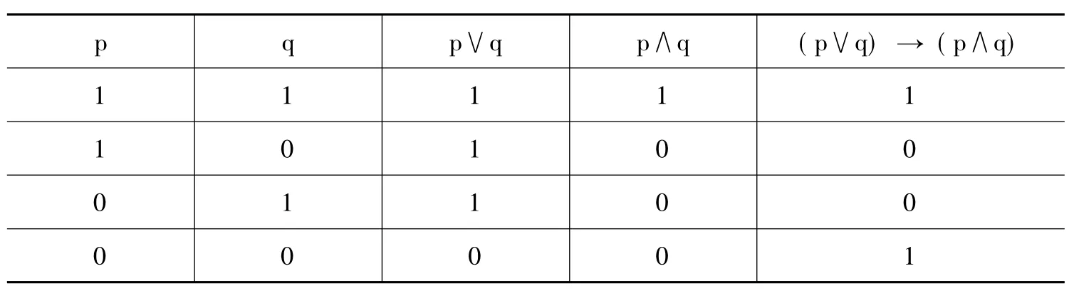


图 3 真值表示例

程序首先检查用户的输入是否存在问题，若输入正确，则将所输入命题公式的命题变元与命题连接词依次提取出来。运算过程中使用了两个栈结构辅助存储中间变量及结果，其中一个是符号栈，存放实时运算的运算符，第二个存放字母对应的真值。程序进行逻辑运算的思路与算术运算大致相同，首先指定&、|、^、~等逻辑运算符的优先级，然后依次进行运算。若当前运算符优先级大于运算符栈顶元素的优先级，则该运算符入栈，若当前运算符优先级小于运算符栈顶元素的优先级，则取真值栈中栈顶的一个元素或两个元素进行运算并将结果再次压入栈中，直到运算符栈为空并且真值栈中只剩下一个元素逻辑运算结束，真值栈中保留的即为运算结果。如此对于每一个变量分别选择T或F进行运算，将命题变元每一种不同排列组合对应的真值求出，即可得到所求的真值表。

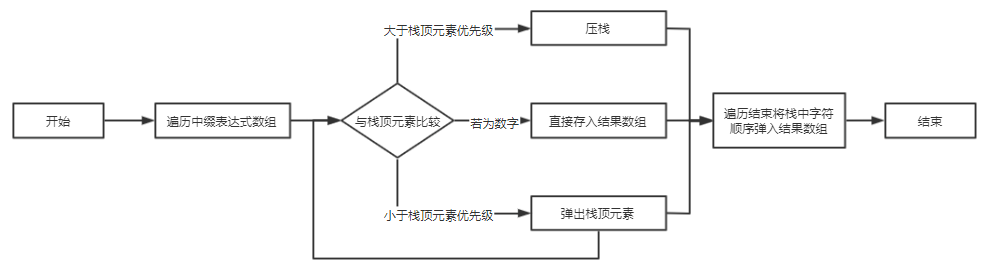


图 4 逻辑运算实现流程图

**2.3 C题：**

主析取范式是指在含有n个命题变元的简单合取式中,若每个命题变元与其否定不同时存在,而两者之一出现一次且仅出现一次,则称该简单合取式为极小项。由若干个不同的极小项组成的析取式称为主析取范式;与A等价的主析取范式称为A的主析取范式。任意含n个命题变元的非永假命题公式A都存在与其等价的主析取范式,并且是惟一的。

主合取范式是在含有n个命题变元的简单析取式中，若每个命题变元与其否定不同时存在，而两者之一出现一次且仅出现一次，称该简单析取式为极大项。由若干个不同的极大项组成的合取式称为主合取范式；与A等价的主合取范式称为A的主合取范式。任意含n个命题变元的非永真命题公式A都存在与其等价的主合取范式，并且是惟一的。

在B题中我们已经求得了命题公式的真值表，因此可以利用命题逻辑的知识简化主析取范式与主合取范式的求解过程。对于主析取范式而言，我们只需求得对应的极小项，即能够使命题公式成立的成真赋值即可，同理对于主合取范式，我们只需求得对应的极大项，即能够使命题公式成立的成假赋值即可，成真赋值与成假赋值都可直接在B题所求的的真值表中查找得到。

1. **数据结构**

**3.1 符号栈与真值栈：**

本项目的核心功能是完成逻辑表达式的计算。由于每个命题变元真值独立，各个元素之间不存在明显的偏序关系或层次关系，也不存在“一对多”与“多对一”的映射，所以可以采用线性结构进行存储。从上文功能分析可知逻辑表达式的求值过程对元素的读取和插入是满足“先进后出”规则的，即栈的“Last In First Out”形式，因此可以建立受限线性表栈对逻辑表达式的计算及中间结果进行存储。

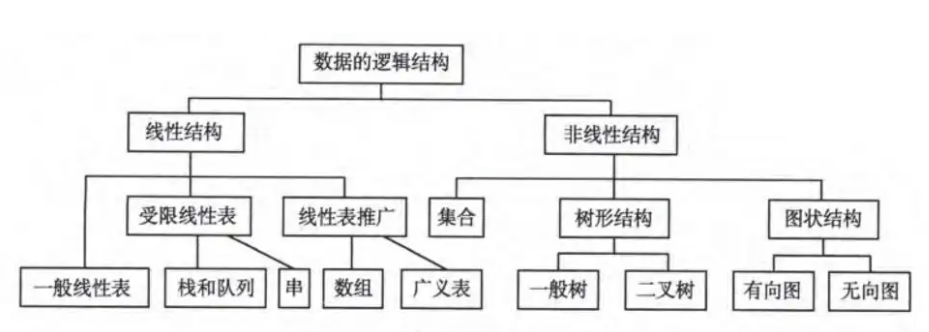
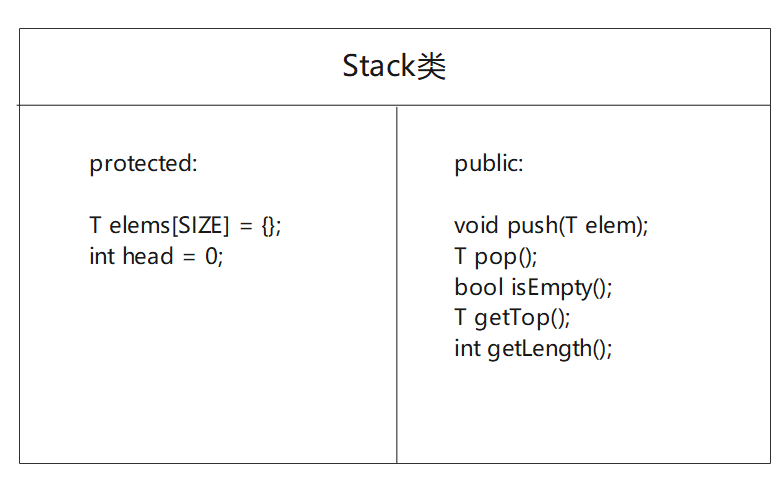


图 5 数据的逻辑结构



**3.2 map集合：**

map是STL的一个关联容器，它提供一对一的[hash](https://so.csdn.net/so/search?q=hash&spm=1001.2101.3001.7020" \t "https://blog.csdn.net/sevenjoin/article/details/_blank)，一般包含两个数据域。第一个称为关键字key，每个关键字只能在map中出现一次，第二个称为该关键字的值value。map以模板即泛型的方式实现，可以存储任意类型的数据，包括使用者自定义的数据类型。map主要用于资料一对一映射的情況，其內部实现自建一颗红黑树，这颗树具有对数据自动排序的功能，在map内部所有的数据都是有序的。本实验共定义了三种map类型，分别是<char,int>、<int,char>、<int,int>，分别命名为Map\_ci、Map\_ic、Map\_ii。下面介绍三种类型map在本程序中的应用。

Map\_ci的关键字为char类型，值为int类型，其主要用于存储运算符的优先级。

上文提到在逻辑表达式的计算过程中若当前运算符优先级大于运算符栈顶元素的优先级，则该运算符入栈，若当前运算符优先级小于运算符栈顶元素的优先级，则进行运算。因此通过Map\_ci快速建立运算符与其优先级间的映射可以提高整个运算过程的执行效率。

priority['('] = 6;

priority[')'] = 6;

priority['!'] = 5;

priority['&'] = 4;

priority['|'] = 3;

priority['^'] = 2;

priority['~'] = 1;

priority['#'] = 0;

Map\_ic的关键字为int类型，值为char类型，其主要用于存储命题变元，将命题变元的序号与命题变元的标识符一一绑定。此过程相当于给每一个命题变元添加了用于识别的id，通过这个id我们可以将命题变元的其他信息与命题变元进行关联，比如命题变元此时的真值、命题变元是否已经被遍历等。

Map\_ii的关键字为int类型，值为int类型，其主要用于记录命题变元的真值，T为1，F为0。假定一个命题变元为“a”,该命题变元在命题公式中第k个出现，则通过上述Map\_ic数据结构我们可以将<k,“a”>绑定，若某一时刻“a”的真值为T，则通过Map\_ii可以将<k，T>绑定，从而记录命题变元此时的真值<“a”，T>。

1. **核心算法**

**4.1 输入处理与命题变元的提取：**

程序首先是对各个输入量的处理，要确定输入的为0或1，否则出错。在一遍扫描输入命题公式的过程中，我们希望能将公式中出现的所有命题变元提取出来，存储在<int，char>类型的map中。为实现此目的，程序定义了一个函数getProposition，该函数可以检查输入公式里面的词法错误，比如是否有非法字符、是否有非字母和公式符号以外的字符出现、是否存在两个相同字母是否连写等。同时getProposition函数还可以检查是否有语法错误，比如二元运算符左右是否存在命题变元等。在遍历命题公式的过程中，程序提取每一个首次出现的命题变元，为之后的逻辑运算以及分析做好预处理。

Map\_ic getProposition(string formula)

{

    Map\_ic proposition;

**int** n\_proposition = 0;

**for** (unsigned **int** i = 0; i < formula.length(); i++)

    {

**char** c = formula[i];

**if** ((c >= 'a' && c <= 'z') || (c >= 'A' && c <= 'Z'))

        {

            //遍历所有命题变项

**int** r = findProposition(proposition, c);

            //cout << r << endl;

**if** (r == -1)

            {

                //说明该命题变项尚未被遍历过

                proposition[n\_proposition] = c;

                n\_proposition++;

            }

        }

**else** **if** (!priority.count(c))

        {

            cout << c << " is undefined!" << endl;

            exit(2);

        }

    }

**return** proposition;

}

**4.2 真值表的求解：**

求解真值表的核心是对命题公式进行逻辑运算，程序遍历命题变元的每一种真值组合，通过符号栈与真值栈计算命题公式的结果，从而组成真值表。函数部分，主要是3个函数，一个为真值表递加函数，通过二进制的加法原理递进产生，一个为分级运算函数，这个函数是通过判断括号，选出最内级括号的内容执行运算函数，这样一级级向外运算，最后得出最终结果，剩下一个为主运算函数，按照运算符号的优先级按顺序进行运算。对于主运算函数，程序进行逻辑运算的思路与算术运算大致相同，首先指定&、|、^、~等逻辑运算符的优先级，然后依次进行运算。若当前运算符优先级大于运算符栈顶元素的优先级，则该运算符入栈，若当前运算符优先级小于运算符栈顶元素的优先级，则取真值栈中栈顶的一个元素或两个元素进行运算并将结果再次压入栈中，直到运算符栈为空并且真值栈中只剩下一个元素逻辑运算结束，真值栈中保留的即为运算结果。通过以上算法，可以实现任意输入公式的真值表计算。

求解命题变项的二进制取值：

Map\_ii toBinary(**int** n\_proposition, **int** index)

{

    Map\_ii result;

**for** (**int** i = 0; i < n\_proposition; i++)

    {

**int** r = index % 2;

        result[n\_proposition - 1 - i] = r;

        index = index / 2;

    }

**return** result;

}

求解指定数字的二次方的值：

**int** pow2(**int** n)

{

**if** (n == 0)

**return** 1;

**else**

**return** 2 \* pow2(n - 1);

}

求解命题变项组合的运算结果：

**int** calculate(string formula, Map\_ic pSet, Map\_ii value)

{

    stack<**char**> opter;

    stack<**int**> pvalue;

    opter.push('#');

    formula = formula + "#";

**for** (unsigned **int** i = 0; i < formula.length(); i++)

    {

**char** c = formula[i];

**if** ((c >= 'a' && c <= 'z') || (c >= 'A' && c <= 'Z'))

        {

            pvalue.push(value[findProposition(pSet, c)]);

        }

**else**

        {

            //此时遍历的是运算符

**char** tmp = opter.top();

**if** (priority[tmp] > priority[c])

            {

**while** (priority[tmp] > priority[c] && tmp != '(')

                {

                    check(pvalue, opter);

                    tmp = opter.top();

**if** (tmp == '#' && c == '#')

                    {

**return** pvalue.top();

                    }

                }

                opter.push(c);

            }

**else**

                opter.push(c);

        }

    }

**return** -1;

}

求解两个命题变项的各种运算结果

**void** check(stack<**int**>& value, stack<**char**>& opter)

{

**int** p, q, result;

**char** opt = opter.top();

**switch** (opt)

    {

**case** '&':

        p = value.top();

        value.pop();

        q = value.top();

        value.pop();

        result = p && q;

        value.push(result);

        opter.pop();

**break**;

**case** '|':

        p = value.top();

        value.pop();

        q = value.top();

        value.pop();

        result = p || q;

        value.push(result);

        opter.pop();

**break**;

**case** '!':

        p = value.top();

        value.pop();

        result = !p;

        value.push(result);

        opter.pop();

**break**;

**case** '^':

        q = value.top();

        value.pop();

        p = value.top();

        value.pop();

        result = !p || q;

        value.push(result);

        opter.pop();

**break**;

**case** '~':

        p = value.top();

        value.pop();

        q = value.top();

        value.pop();

        result = (!p || q) && (p || !q);

        value.push(result);

        opter.pop();

**break**;

**case** '#':

**break**;

**case** '(':

**break**;

**case** ')':

        opter.pop();

**while** (opter.top() != '(')

        {

            check(value, opter);

        }

**if** (opter.top() == '(')

        {

            opter.pop();

        }

**break**;

**default**:

**break**;

    }

}

1. **代码改进**

对于提供的源代码仍有部分不完善之处。比如程序中有warning存在，以及考虑不够全面，很多地方没有进行错误处理。为此我修复了程序中的问题，并完善了错误处理部分，以提高代码的鲁棒性。

**5.1 逻辑运算与位运算：**

实现逻辑运算可以通过逻辑运算符与位运算符两种方式实现，比如当非布尔型数值取0或1时，按位或|的运算结果与逻辑运算||的结果是一致的。但当使用逻辑运算方式时，函数处理对象应当定义为bool类型。虽然非布尔型数值取0或1在功能上是等价的，但对于计算机而言存在风险，因为如果人为给非布尔类型变量赋值时超过了0和1的范围，程序有出错的风险。因此，我将程序中所有待进行逻辑运算的变量全部改为了布尔类型。



**5.2 size\_t与int类型：**

# int 类型一般小于等于数据线的宽度，而 size\_t 类型一般为大于等于地址线宽度，地址线宽度通常是大于数据线宽度的。比如在数据只有8位的年代，地址率先进入10 位、12 位；在数据16位的年代，地址也已经进入了20位、24 位；后来 int 普遍是 32 位，而地址线宽度在主流平台中都是64位。所以，无论int还是unsigned int都很可能小于地址线宽度需要的大小，进而就必须使用size\_t数据类型。当size\_t类型变量转换为int时可能会发生数据的丢失，因此应当避免size\_t数据类型强转int。

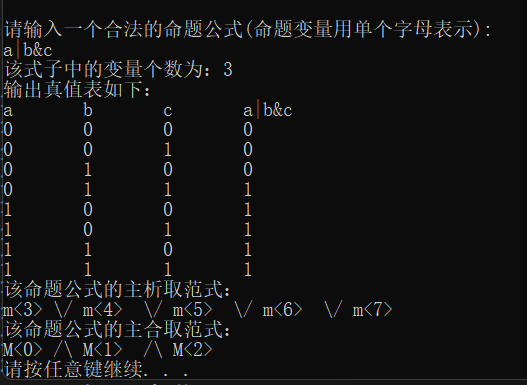


**5.3 错误处理：**

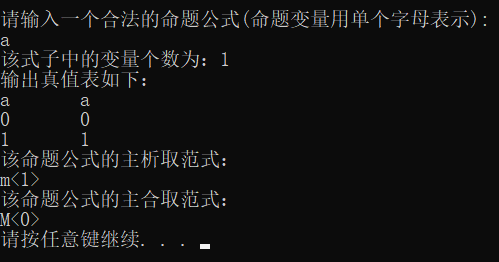
本项目对项目错误处理的修改主要在calculate函数里。在原先的求解过程中，程序分别检查输入表达式的词法错误与语法错误，这与编译原理对语句的词法语法处理方法非常类似。程序运用了符号栈与真值栈求解命题公式真值，并且添加了“#”辅助判定，通过栈外运算符与栈顶运算符的优先级大小关系确定程序此时应当执行的动作。但原先程序无法检查出连续输入多个命题变项、括号不匹配等情况。因此在符号栈与真值栈运算完成后，需要判别一下栈内是否仅存在一个元素。若元素个数超过一个，则程序报错，否则栈内元素即为最终答案。同时原先代码对仅有一个变量的情况不能很好判别，这里对此类情况进行了改进。

1. **实验测试**

**6.1 常规测试：**

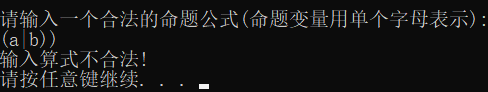


**6.2 边界测试-只有一个命题变量：**

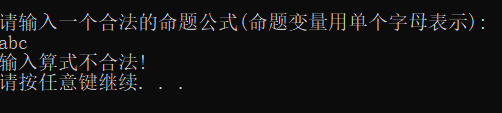


**6.3 错误测试：**

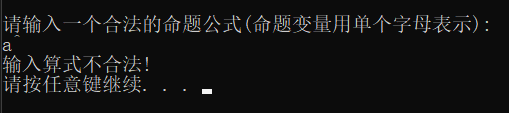
括号不匹配：



连续输入多个变量：



命题公式不完整：



1. **心得体会**

通过本次实验，我进一步理解了命题逻辑中的联接词、真值表、主范式等知识，并学会了如何使用它们来解决实际问题。在课堂的理论学习中，我掌握了几种常见的命题逻辑联接词与命题公式的计算，学会了如何求一个给定命题公式的真值表、求一个给定命题公式的主析取范式与主合取范式。但作为软件工程专业的学生，仅仅了解理论知识还不够，更重要的是明白如何将离散数学的各项理论应用于程序开发、算法设计等方面上，真正将计算机知识与数学知识融合。

在A题的求解过程中，我熟悉了C语言本身支持的与&&、或||、非!三种逻辑运算符，并认识了非布尔类型变量也可通过位运算实现逻辑运算。在B题的求解过程中，程序对输入的处理非常精妙，不仅能识别词法错误，还可识别语法错误，这与编译原理对语句的词法语法处理方法非常类似。程序运用了符号栈与真值栈求解命题公式真值，并且添加了“#”辅助判定，通过栈外运算符与栈顶运算符的优先级大小关系确定程序此时应当执行的动作，这点与算术运算的实现方法是非常类似的。在C题的求解过程中，我进一步理解了范式的含义以及如何通过查表求得命题公式的成真赋值、成假赋值、极大项、极小项、主析取范式、主合取范式以及它们之间的相互联系。

除此之外，本实验也教会了我如何提高程序的鲁棒性，这点对于每一位程序设计工作者都是十分重要的。