# 1、解题思路

1.1、需求分析：

求任意一个命题公式的真值表（B），并根据真值表求主范式（C）。

根据这个需求，首先要根据输入的命题公式得出其命题变项，再求出所有赋值的结果得到其真值表，最后根据真值表计算出的值求出主范式。

1.2、实验原理：  
（1）真值表:

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。

### （2）主范式：

a、主析取范式：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。

b、主合取范式：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。

## 1.3、设计思路：

### ①、给定命题变项的值，求表达式的值：

相当于求一个中缀表达式的值，可以使用栈来求解。具体操作见核心算法部分。

### ②、模拟将各组真值指派给合式公式，求出所有的赋值结果：

可以用一个二进制数来表示每一组真值的各个数位，在二进制数上，每一位上的1和0对应相应的命题变量的真和假。总共存在（n表示命题变项的个数）种指派方式，可以使用一个十进制计数器i来进行模拟，从0开始不断加一，直到。

### ③、求主范式：

所有成真赋值都是极小项，成假赋值都是极大项，故将所有的成真赋值时计数器的值计入主析取范式，而将所有成假赋值时计数器的值计入主合取范式。如果是重言式，即无成假赋值，故没有主和取范式；若是矛盾式，则没有成真赋值，故无主析取范式。

# 2、数据结构：

在该实验中，需要用到多种数据结构：

## 、存储命题变项的值：

使用STL中的vector。

vector的逻辑结构是线性结构，存储结构是可以进行动态内存申请的顺序表。使用该存储结构的好处是操作方便，同时根据需求，事先并不知道命题变项的个数，通过vector动态的内存申请，可以节省内存。

## 、存储命题变项在vector中的索引：

采用静态数组。

静态数组也是线性结构，存储结构也是顺序表。由于已知命题变项为单个的ascii码，所以只存在少于128种情况，可以用一个静态数组来进行存储。使用该数据结构的好处是查取方便，查询只需要O(1)的时间复杂度。由于查询命题变项在vector中的索引操作在程序中很常见，故降低其时间复杂度对于降低程序的时间复杂度很有帮助。

## 、计算赋值结果时存储操作数和操作符：

使用STL中的stack。

stack的逻辑结构也是线性结构。由于功能受限，有着FILO的特点，经常利用该数据结构计算中缀表达式的值。

# 3、核心算法：

利用栈模拟计算中缀表达式的值：

使用两个操作栈——OPTR（储存操作符），OPND（储存操作数）。先将#压入OPTR中，再依次遍历表达式，如果遇见操作数（真值），则将该操作数压入OPND；如果遇见操作符：

<1>栈顶操作符优先级低于新入栈操作符，将新操作符入栈

<2>新入栈操作符优先级低于栈顶操作符，取出栈顶操作符，与OPND中的栈顶两个（或一个，计算 ! 时），操作数进行运算，并将结果入栈。

<3>栈顶操作符优先级等于新入栈操作符，将栈顶操作符弹出，新操作符也不入栈。

当表达式遍历结束后，将#入栈。

至于操作符的优先级，可以用一张优先级表进行表示，该表如下：

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 新入栈  栈顶 | ~ | ^ | | | & | ! | ( | ) | # |
| ~ | > | < | < | < | < | < | > | > |
| ^ | > | > | < | < | < | < | > | > |
| | | > | > | > | < | < | < | > | > |
| & | > | > | > | > | < | < | > | > |
| ! | > | > | > | > | > | < | > | > |
| ( | < | < | < | < | < | < | = | NULL |
| ) | > | > | > | > | > | NULL | > | > |
| # | < | < | < | < | < | < | NULL | = |

在表中可以发现，在栈顶时，除了左括号和#号外的运算符优先级都高于新入栈的自己；而左括号运算符在作为新入栈元素时优先级最高，而在作为栈顶元素时优先级最低；右括号恰好相反——在作为新入栈元素时优先级最低，而在作为栈顶元素时优先级最高。

而左右括号，左右＃之间的运算级相等。

这样设置的目的是，在同级别的运算符欲进栈时，可以先取出栈内运算符进行计算；而左括号作为新入栈元素时优先级最高，不管栈顶元素为何（除了右括号，逻辑错误），都能接受左括号入栈，因为括号内的部分要优先计算，由于栈有着FILO的性质，栈顶元素先计算，所以括号内部分先计算；而左括号作为栈顶元素时优先级最低，不论新入栈元素为何（除了#，逻辑错误，以及），互相消除），都能接受入栈，从而进行计算。而右括号在作为新入栈元素时优先级最低（除了左括号，相互消除，和#，逻辑错误），不管栈顶元素为何，都无法接受右括号入栈，只能先弹出栈顶元素进行计算，直到遇见左括号相互消除为止，这样就完成了括号内表达式的计算。

# 4、程序测试：

## a、基本测试：

1、测试输入：a^b~c|b&a^c

实验预期：

命题变项的个数为：3

真值表：

a b c a^b~c|b&a^c

0 0 0 1

0 0 1 1

0 1 0 1

0 1 1 1

1 0 0 0

1 0 1 0

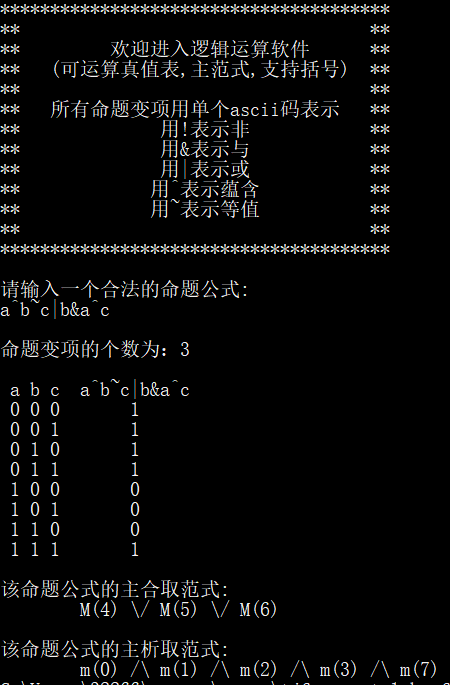
1 1 0 0

1 1 1 1

主合取范式: M(4) \/ M(5) \/ M(6)

主析取范式:m(0) /\ m(1) /\ m(2) /\ m(3) /\ m(7)

实验结果：



2、测试输入：!a^b~(c|d~(!b&c)^!d)|a

真值表：

a b c d !a^b~(c|d~(!b&c)^!d)|a

0 0 0 0 1

0 0 0 1 0

0 0 1 0 0

0 0 1 1 1

0 1 0 0 0

0 1 0 1 1

0 1 1 0 1

0 1 1 1 1

1 0 0 0 1

1 0 0 1 1

1 0 1 0 1

1 0 1 1 1

1 1 0 0 1

1 1 0 1 1

1 1 1 0 1

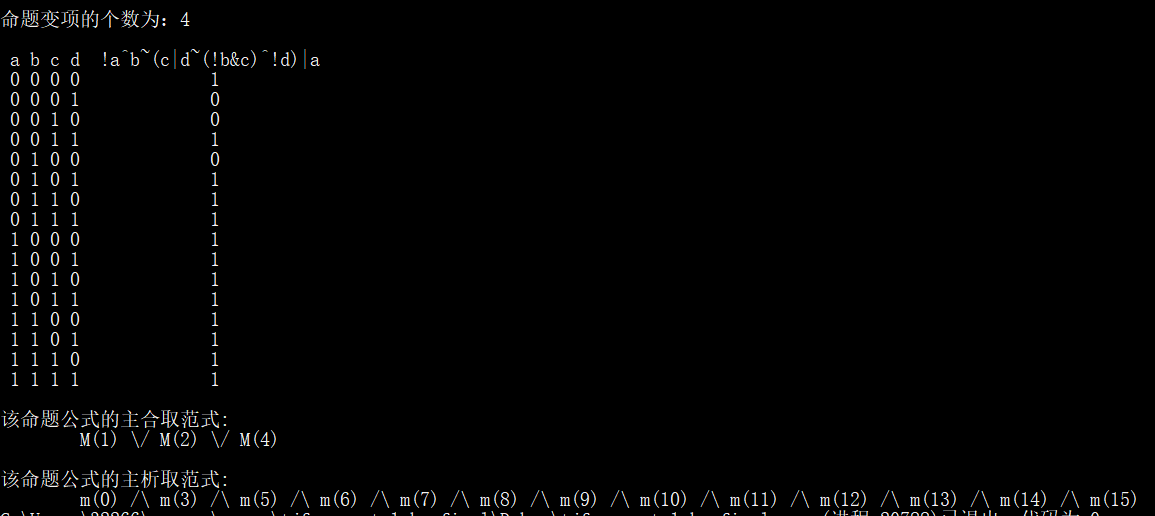
1 1 1 1 1

主合取范式: M(1) \/ M(2) \/ M(4)

主析取范式:

m(0) /\ m(3) /\ m(5) /\ m(6) /\ m(7) /\ m(8) /\ m(9) /\ m(10) /\ m(11) /\ m(12) /\ m(13) /\ m(14) /\ m(15)

实验结果：

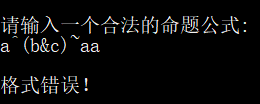


b、出错测试：

1、测试输入：a^(b&c)~aa

预期：错误提示

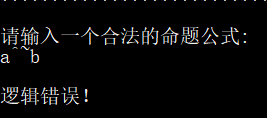
实验结果：



2、测试输入：a^~b

预期：错误提示

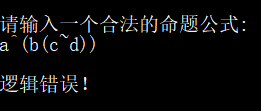
实验结果：



3、测试输入：a^(b(c~d))

预期：错误提示

实验结果：



4、测试输入：a|(b^c&d))

预期：错误提示

实验结果：

