

## 

**《离散数学》课程实验报告**

# 题目 命题逻辑联接词、主范式、真值表

## 姓 名 尹诚成

学 号 2351279

学 院 计算机科学与技术学院

专 业 软件工程

教 师 唐剑锋

二〇二四 年 十二 月 二十 日

目录

1 实验目的1

2 实验内容1

2.1 命题逻辑联结词1

2.2 真值表、主范式1

3 实验环境2

4 实验原理2

4.1 合取2

4.2 析取2

4.3 条件3

4.4 双向条件3

4.5 真值表3

4.6 主合取范式4

4.7 主析取范式4

5 实验过程（命题逻辑联接词）4

5.1 实验思路4

5.2 实验设计5

5.2.1 数据结构设计5

5.2.2 结构体与类设计5

5.2.3 程序主题架构设计5

5.3 程序功能实现6

5.3.1 逻辑值输入功能的实现 6

5.3.2 退出程序功能的实现6

5.4 核心算法实现7

6 实验过程（真值表、主范式）7

6.1 实验思路7

6.2 实验设计7

6.2.1 数据结构设计7

6.2.2 结构体与类设计7

6.2.3 程序主体架构设计8

6.3 程序功能实现8

6.3.1 命题公式合法性检验功能8

6.3.2 命题公式输出功能11

6.3.3 命题公式变量提取功能11

6.3.4 真值表输出功能12

6.3.5 主合取范式输出功能13

6.3.6 主析取范式输出功能13

6.3.7 退出程序功能的实现14

7 实验数据分析14

7.1 命题逻辑联接词14

7.2 真值表、主范式16

8 实验心得23

9 程序源文件24

9.1 命题逻辑联接词24

9.2 真值表、主范式25

1. **实验目的**

本实验课程训练学生掌握命题逻辑中的联接词、真值表、主范式等，进一步能用它们来解决实际问题。通过实验提高学生编写实验报告、总结实验结果的能力；使学生具备程序设计的思想，能够独立完成简单的算法设计和分析。

1. **实验内容**
   1. 命题逻辑联接词

本实验要求从求两个命题变元P和Q的真值，并通过程序计算它们的合取(∧)、析取（∨）、条件（→）和双向条件（↔）的真值。用户输入两个命题变元P和Q的值，程序计算出这些联接词的结果。

合取（∧）：当且仅当P和Q都为真时，P∧Q为真。

析取（∨）：如何P和Q中至少有一个为真，P∨Q为真。

条件（→）：如果P为假或P和Q都为真，P→Q为真。

双向条件（↔）：当P和Q有相同的真值时，P↔Q为真。

* 1. 真值表、主范式

本实验要求求任意一个命题公式的真值表，并根据真值表求出公式的主析取范式(DNF)和主合取范式(CNF)。用户输入一个命题逻辑公式，程序先判断该公式是否有效，然后计算并显示其真值表，并基于真值表计算命题逻辑公式的主析取范式和主合取范式。

真值表：对于给定的命题逻辑公式，真值表展示了在所有可能的命题变元真 值组合下，公式的真值。

主析取范式（DNF）：它是一种逻辑公式的标准化形式，由若干个极小项的析 取（逻辑“或”）组成。极小项是一个只包含原子命题或其否定的合取（逻辑 “与”）表达式，且每个变量在极小项中只出现一次。

主合取范式（CNF）：与 DNF类似，CNF是由若干个极大项的合取（逻辑“与”） 组成。极大项是一个只包含原子命题或其否定的析取（逻辑“或”）表达式，且每个变量在极大项中只出现一次。

1. **实验环境**

程序开发语言：C++

集成开发环境：Microsoft Visual Studio 2022 (Release 模式)

编译运行环境：本项目适用于x86架构和x64架构

1. **实验原理**
   1. 合取

合取是二元命题联结词。将两个命题P、Q联结起来，构成一个新的命题P∧Q, 读作P、Q的合取, 也可读作P与Q。P、Q的真值间的关系为：当且仅当两个命题都为真时，它们的合取才为真。如果其中任何一个命题为假，它们的合取就为假。

表 4.1 合取运算真值表

|  |  |  |
| --- | --- | --- |
| P | Q | P∧Q |
| 0 | 0 | 0 |
| 0 | 1 | 0 |
| 1 | 0 | 0 |
| 1 | 1 | 1 |

* 1. 析取

析取是二元命题联结词。将两个命题P、Q联结起来，构成一个新的命题P∨Q, 读作P、Q的析取, 也可读作P或Q。P、Q的真值间的关系为：当至少一个命题为真时，它们的析取 就为真。只有当两个命题都为假时，它们的析取才为假。

表 4. 2 析取运算真值表

| P | Q | P∨Q |
| --- | --- | --- |
| 0 | 0 | 0 |
| 0 | 1 | 1 |
| 1 | 0 | 1 |
| 1 | 1 | 1 |

* 1. 条件

条件是二元命题联结词。将两个命题P、Q联结起来，构成一个新的命题P→Q, 读作P条件Q, 也可读作如果P，那么Q。P、Q间的真值关系为：如果第一个命题为真，则第二个命题也必须为真，整个条件才为真。唯一使条件为假的情况是P为真而Q为假。

表 4. 3 合取运算真值表

|  |  |  |
| --- | --- | --- |
| P | Q | P→Q |
| 0 | 0 | 1 |
| 0 | 1 | 1 |
| 1 | 0 | 0 |
| 1 | 1 | 1 |

* 1. 双向条件

双向条件是二元命题联结词。将两个命题P、Q联结起来，构成一个新的命题P↔Q, 读作P双条件于Q，也可读作P等价Q。P、Q间的真值关系为：

当两个命题具有相同的真值时，它们的双向条件为真

表 4. 4 合取运算真值表

|  |  |  |
| --- | --- | --- |
| P | Q | P↔Q |
| 0 | 0 | 1 |
| 0 | 1 | 0 |
| 1 | 0 | 0 |
| 1 | 1 | 1 |

* 1. 真值表

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。

* 1. 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，则称该简单析取式为极大项。主合取范式是通过对极大项进行合取（即逻辑“与”操作）来构造的。换句话说，它是由若干个不同的极大项通过逻辑“与”连接起来的形式。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. 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。主析取范式是通过对极小项进行析取（即逻辑“或”操作）来构造的。换句话说它是由若干个不同的极小项通过逻辑“或”连接起来的形式。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。

1. **实验过程（命题逻辑联结词）**
   1. 实验思路

本实验要求设计一个程序来演示命题逻辑联结词的功能。实验通过用户输入两个命题变元P和Q的值，程序计算出它们的合取(∧)、析取（∨）、条件（→）和双向条件（↔）的真值。

* 1. 实验设计
     1. 数据结构设计

本程序主要设计基本的逻辑运算，因此不需要复杂的数据结构。主要使用基本的数据类型如布尔值来存储命题变元的真值。

* + 1. 结构体与类设计

在这个实验中，没有使用结构体或类，因为程序的规模和复杂性不需要这样的封装。

* + 1. 程序主体架构设计

程序从主函数main开始。程序首先清理屏幕，然后打印界面和输入说明，通过函数 inputLogicalValue 获取用户输入的命题变元P和Q的值（0/1），最后计算并显示这些命题变元经过不同联接词处理的结果，最后询问用户是否退出。

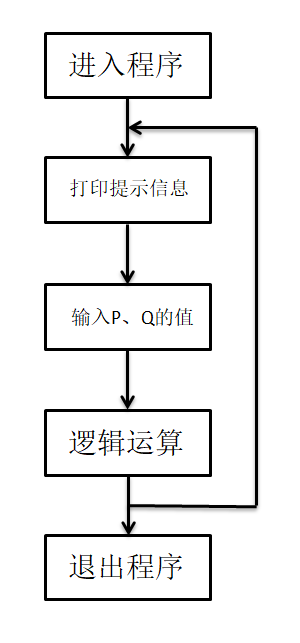


图5.2.3 程序主题架构流程设计图

* 1. 程序实现功能
     1. 逻辑值输入功能的实现

函数 inputLogicalValue 用于输入逻辑值，处理二进制（真/假）值的用户输入，这里分别用'0'和‘1’或者通过参数 FalseValue 和 TrueValue 指定的其他字符表示。

程序功能实现思路：

(1)FalseValue 表示假的值，默认值为'0'。TrueValue表示真的值，默认值为 '1'；

(2)该函数使用一个无限循环（while(true)），等待用户输入。使用\_getch()函数从键盘获取一个字符，而不回显到控制台。首先判断是否是特殊键（如回车键），这些键的ASCII值通常为0或-32。如果是，则再次调用\_getch()来获取实际的键值；

(3)接下来判断输入的字符是否与falseValue或trueValue相匹配。如果匹配，将输入的字符打印到控制台，并根据输入的值返回false或true。

* + 1. 退出程序功能的实现

退出程序的功能是通过在main函数内部的一个循环结构实现的。此功能允许用户在完成操作后选择是否退出程序。

程序功能实现思路：

(1)程序使用 do-while 循环来反复执行程序和询问用户是否退出程序。在每次循环的开始，使用system("cls")清除屏幕，以提供清晰的界面；

(2)在完成一轮操作后，程序会输出询问用户是否退出程序的提示：“是否退出程序 [y/n]: ”。此时，程序再次调用inputLogicalValue函数，这次用于接收用户的退出决定。参数被设置为'n'（不退出）和'y'（退出）；

(3)如果用户输入'y'（对应真值），则inputLogicalValue函数返回true， 导致 do-while 循环结束，程序随之退出。如果用户输入'n'（对应假值），inputLogicalValue 函数返回 false，do-while 循环继续，用户可以进行新一轮操作。

* 1. 核心算法实现

合取（∧）：当且仅当P和Q都为真时，P∧Q为真。通过代码 p && q表示。

析取（∨）：如何P和Q中至少有一个为真，P∨Q为真。通过代码 p || q 表示。

条件（→）：如果P为假或P和Q都为真，P→Q为真。通过代码 !p || q 表示。

双向条件（↔）：当P和Q有相同的真值时，P↔Q为真。通过代码 (!p || q) && (!q || p) 表示。

1. **实验过程（真值表、主范式）**
   1. 实验思路

本实验的目标是设计一个程序，用于生成任意命题逻辑公式的真值表，并基于该真值表计算出公式的主析取范式（DNF）和主合取范式（CNF）

* 1. 实验设计
     1. 数据结构设计

std::set<char>：用于存储命题逻辑公式中的唯一变量。

std::map<char,bool>：映射每个变量到其对应的真值。

std::stack<char>和 std::stack<bool>：用于后缀表达式的解析和计算。

* + 1. 结构体与类设计

本实验未使用结构体或类，因为所需的数据结构和操作可以通过标准库容器和函数直接实现。

* + 1. 程序主体架构设计

程序从主函数main开始，包括用户界面的呈现、命题逻辑表达式的输入和有效性检查、真值表的生成和主范式的导出。

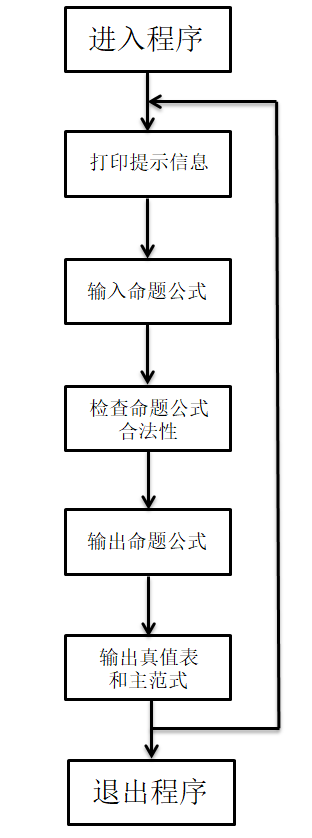


图6.2.3 程序主题架构设计流程图

* 1. 程序功能实现
     1. 命题公式合法性检验功能

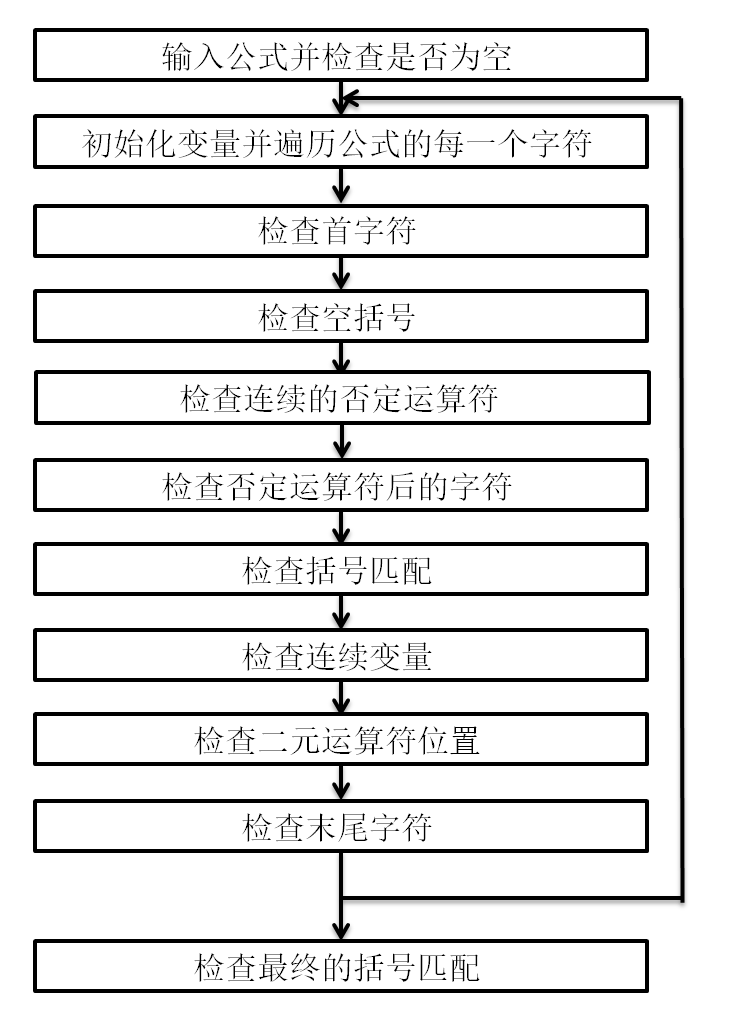


图6.3.1 命题公式合法性检验功能实现流程图

函数checkExpression检查多种语法规则，如仅允许特定字符（字母、 逻辑运算符和括号）、运算符的正确放置、括号的正确嵌套和避免空括号。确保二元运算符不被不当使用（例如，不应在表达式的开始或结束处使用，也不应在没有操作数的情况下连续使用）。检查否定运算符的正确放置（不直接跟在字母 后）和不连续出现。该函数还检查单字符变量，确保不使用多字符变量。此功能关键在于确保公式格式正确，无语法错误，从而保障后续操作的准确性。

命题公式输入要求：

1. 字符'!'表示非
2. 字符'&'表示与
3. 字符'|'表示或
4. 字符'^'表示蕴含
5. 字符'~'表示等值
6. 命题公式中只能输入大小写字母、括号和以上五种运算符，空格后的内容将被截断
7. 命题公式中的括号嵌套匹配
8. 命题公式仅适用于单字符变量，不适用于多字符变量
9. 命题公式中每个运算符前后必须连接变量

程序实现思路：

(1)检查空公式：判断输入的公式是否为空。如果为空，向用户提示命题公式为空，请重新输入！”并返回false，表示公式不合法；

(2)初始化变量：使用 std::stack<char>类型的 parentheses 来跟踪公 式中的括号匹配情况，使用prev\_char来存储前一个字符，初始值为'\0'；

(3)遍历公式中的每个字符，检查每个字符是否合法：公式中的字符必须是字母、逻辑运算符（通过函数isOperator检查）或括号。如果不是，提示“输入错误：命题公式含有非法字符，请重新输入！”并返回false；

(4)检查首字符：若公式首字符是二元运算符（'&','|','~','^'），则提示“输入错误：命题公式第一个字符不能为二元运算符，请重新输入！”并返回false；

(5)检查空括号：如果出现一对空括号（即连续的'('和')'），提示“输入错误：命题公式不能出现空括号，请重新输入！”并返回false；

(6)检查连续的否定运算符：如果发现连续的'!'字符，提示“输入错误：命题公式不能出现连续取非操作，请重新输入！”并返回false；

(7)检查否定运算符后的字符：如果'!'前紧跟字母，提示“输入错误：命题公式中取非操作后必须连接变量，请重新输入！”并返回false；

(8)检查变量和括号的连接：如果字母紧跟')'或'('紧跟字母，提示“命题 公式中变量与括号的连接不正确，请重新输入！”并返回false；

(9)括号匹配：使用栈parentheses来检查括号是否匹配。遇到'('时压入 栈，遇到')'时弹出栈顶元素。如果在')'出现时栈为空，提示“输入错误：命题公式中括号不匹配，请重新输入！”并返回false；

(10)检查变量连续性：如果连续出现两个字母，提示“输入错误：命题公式中变量不能连续出现，请重新输入！”并返回false；

(11)检查二元运算符的位置：如果'&'、'|'、'~'或'^'运算符前面不是字母或')'，提示“输入错误：命题公式中每个二元运算符前后必须连接变量，请重新输入！”并返回false；

(12)检查末尾字符：如果公式以运算符结尾，提示“输入错误：命题公式不能以运算符结尾，请重新输入！”并返回false；

(13)检查最终的括号匹配：最后检查parentheses栈是否为空。不为空表示括号不匹配，提示“输入错误：命题公式中括号不匹配，请重新输入！”并返回false。

* + 1. 命题公式输出功能

函数 replaceLogicalOperators 将公式中的特定符号（如"&","|","~","^"）替换为更具可读性的符号（如"∧","∨","←→","→"）。这样可以提高公式的可读性，并更接近数学逻辑中常用的表示方法。

* + 1. 命题公式变量提取功能

函数countVariables遍历公式中的每个字符，识别并收集字母字符（即 变量）。它使用一个集合（std::set<char>）来存储变量，保证每个变量只被记录一次。

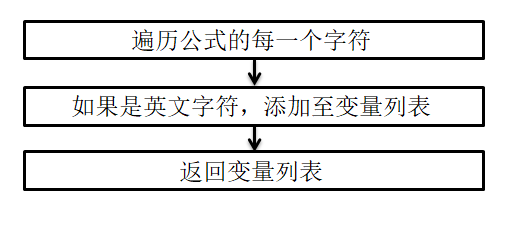


图6.3.3 命题公式变量提取功能实现流程图

* + 1. 真值表输出功能

真值表输出功能是命题逻辑分析中的核心部分，用于展示一个命题公式在不 同变量赋值下的真值结果。这一功能涉及到对命题公式的逻辑解析、变量赋值的 枚举，以及结果的展示。 程序功能实现思路：

(1)初始化和变量提取：首先，通过countVariables函数从输入的命题 公式中提取所有唯一的变量。这些变量被存储在一个集合中，以保证它们的唯一性；

(2)打印表头：打印表头的头部分隔线。打印表头，其中包括每个变量的名称和最终的“Value”列，用于展示公式在特定变量赋值下的计算结果；

(3)枚举变量赋值：对于variableCount个变量，存在2^ variableCount种可能的真值赋值组合。遍历这些组合，每个组合对应真值表的一行。对于每个组合，通过位运算设置每个变量的值（真或假），并打印这些值作为真值表的一部分；

(4)计算公式的真值：使用函数 calculatePostfixExpression 计算中缀表达式的结果，在函数 calculatePostfixExpression 中创建一个布尔值堆栈values来存储每个变量的真值。遍历表达式的每个字符，对于字符变量则将字符变量的真值堆入栈中；对于单目运算符’!’，则弹出栈顶的字符变量的真值，取反后重新堆入栈中；对于双目运算符，则弹出栈顶的两个字符变量的真值，通过函数calculateVariable计算它们的值，重新堆入栈中，遍历结束后返回栈顶元素的真值。

(5)计算并输出结果：在遍历过程中，将每个赋值组合下的计算结果输出，打印每个赋值组合下公式的计算结果。

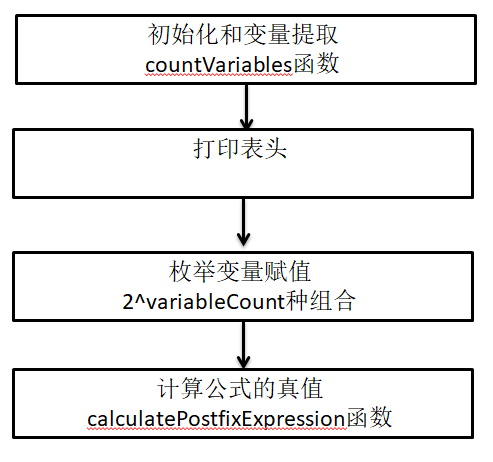


图6.3.4 真值表输出功能实现流程图

* + 1. 主合取范式输出功能

此功能用于输出命题逻辑表达式的主合取范式和对应的成假赋值。主合取范式提供了一种使用“与”（合取）操作符连接若干子句的方式来表示命题公式，而每个子句是一些变量或其否定的“或”（析取）组合。

程序功能实现思路：

(1)计算和存储公式结果：在计算真值表的过程中，每个变量赋值组合下公式的计算结果被存储在一个conjunctionParadigm向量中。该向量存储了主合取范式的成假赋值（已从二进制转化为十进制）；

(2)输出主合取范式：遍历conjunctionParadigm向量，输入其中的成假赋值。

* + 1. 主析取范式输出功能

主析取范式与成真赋值的输出功能与主合取范式与成真赋值的输出功能类似。

此功能用于输出命题逻辑表达式的主析取范式和对应的成真赋值。主析取范式提供了一种使用“或”（析取）操作符连接若干子句的方式来表示命题公式，而每个子句是一些变量或其否定的“与”（合取）组合。

程序功能实现思路：

(1)计算和存储公式结果：在计算真值表的过程中，每个变量赋值组合下公式的计算结果被存储在一个analyticParadigm向量中。该向量存储了主合取范式的成假赋值（已从二进制转化为十进制）；

(2)输出主合取范式：遍历analyticParadigm向量，输入其中的成假赋值。

* + 1. 退出程序功能实现

同5.3.2退出程序功能的实现。

1. **实验数据分析**
   1. 命题逻辑联结词

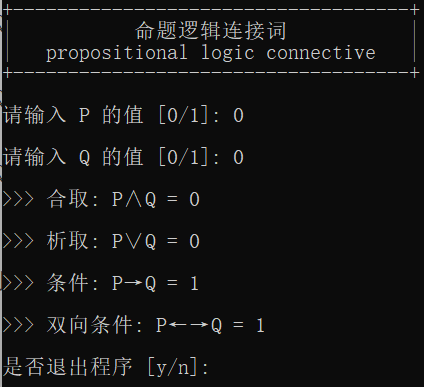


图7.1.1 实验数据1

实验数据分析1：当P的逻辑值为0，Q的逻辑值为0时，P∧Q为假，P∨Q 为假，P→Q为真，P↔Q为真。

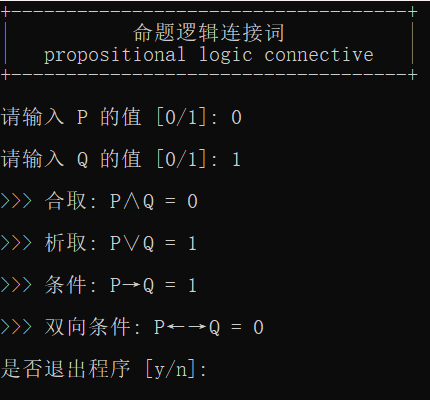


图7.1.2 实验数据2

实验数据分析2：当P的逻辑值为0，Q的逻辑值为1时，P∧Q为假，P∨Q 为真，P→Q为真，P↔Q为假。

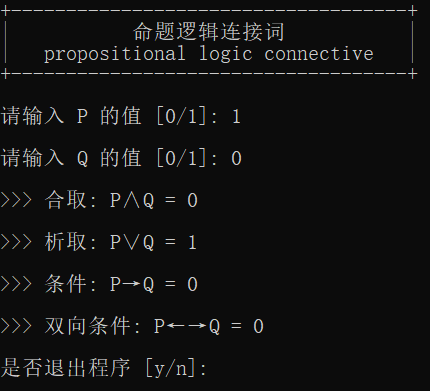


图7.1.3 实验数据3

实验数据分析3：当P的逻辑值为1，Q的逻辑值为0时，P∧Q为假，P∨Q 为真，P→Q为假，P↔Q为假。

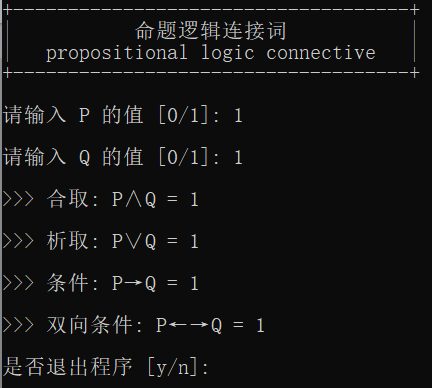


图7.1.4 实验数据4

实验数据分析4：当P的逻辑值为1，Q的逻辑值为1时，P∧Q为真，P∨Q 为真，P→Q为真，P↔Q为真。

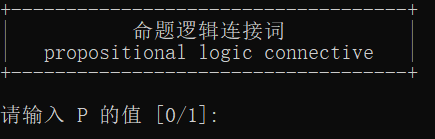


图7.1.5 实验数据5

实验数据分析5：当输入n时，程序清空屏幕并执行下一次命题逻辑联接词 运算。在输入P和Q的逻辑值时只能输入字符0或1，若输入其他字符则不回显，光标闪烁直到输入字符0或1。

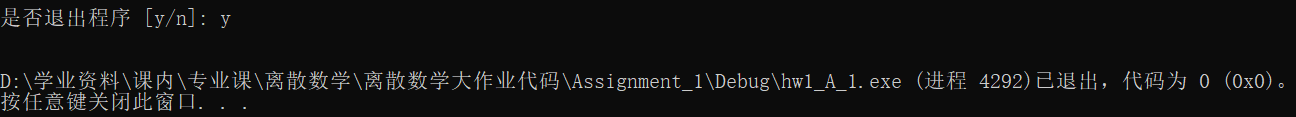


图7.1.6 实验数据6

实验数据分析6：当输入y时，程序退出。

* 1. 真值表、主范式



图7.2.1 实验数据1

实验数据分析1：本实验用于测试非运算符的正确性。 输入命题公式为“!a”，变量个数为1，真值表输出正确，主合取范式为M<1>，主析取范式为m<0>。

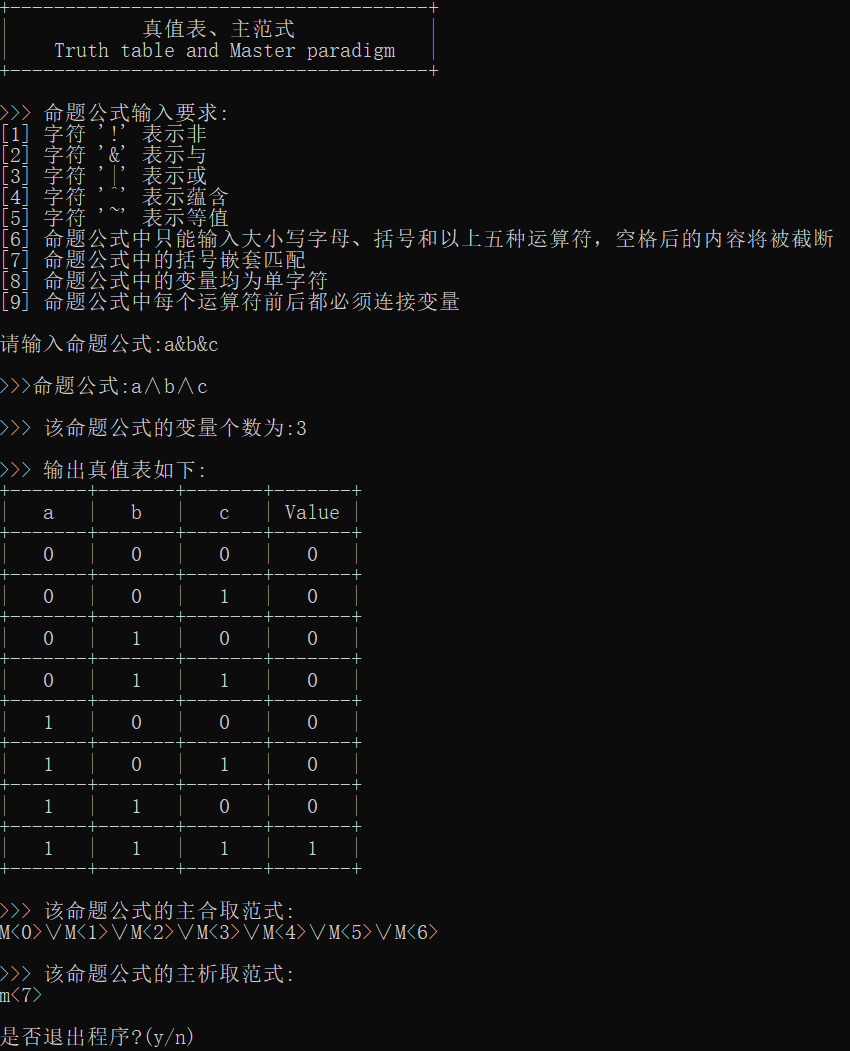


图7.2.2 实验数据2

实验数据分析2：本实验用于测试与运算符的正确性。 输入命题公式为“a∧b∧c”，变量个数为3，真值表输出正确，主合取范式 为M<0>∧M<1>∧M<2>∧M<3>∧M<4>∧M<5>∧M<6>，主析取范式为m<7>。

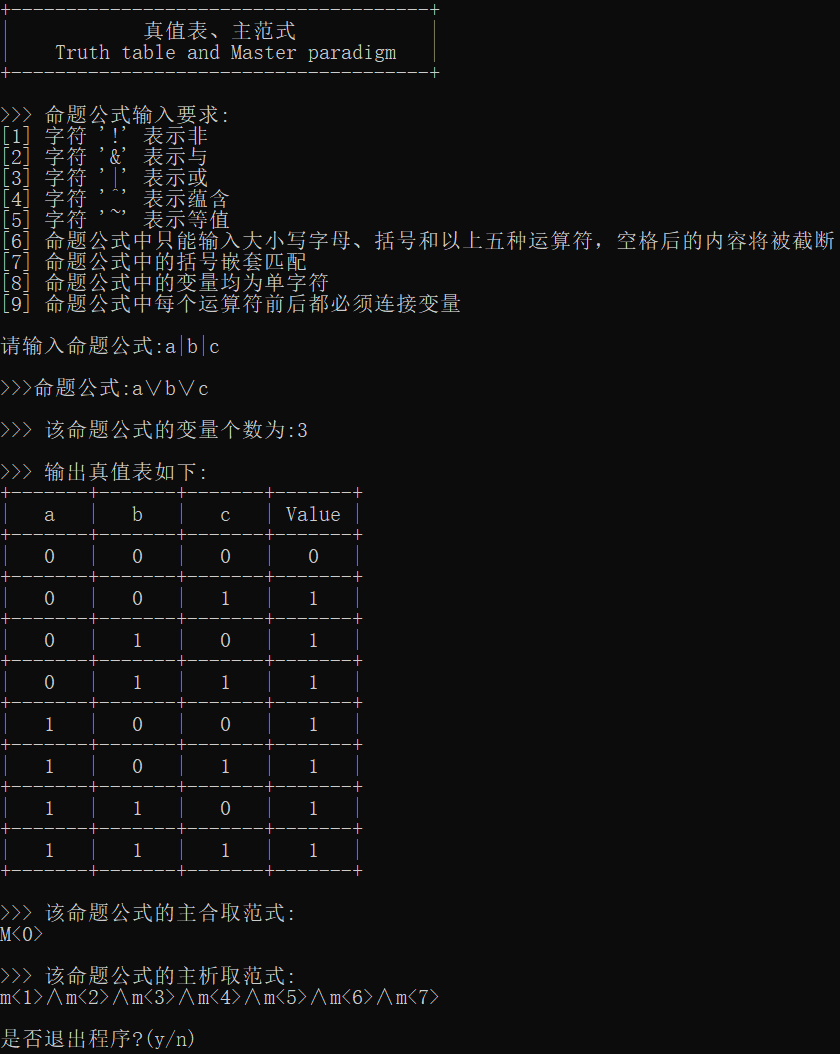


图7.2.3 实验数据3

实验数据分析3：本实验用于测试或运算符的正确性。 输入命题公式为“a∨b∨c”，变量个数为3，真值表输出正确，主合取范式为 M<0> ，成假赋值为000 ， 主 析 取 范 式 为 m<1>∨m<2>∨m<3>∨m<4>∨m<5>∨m<6>∨m<7>。

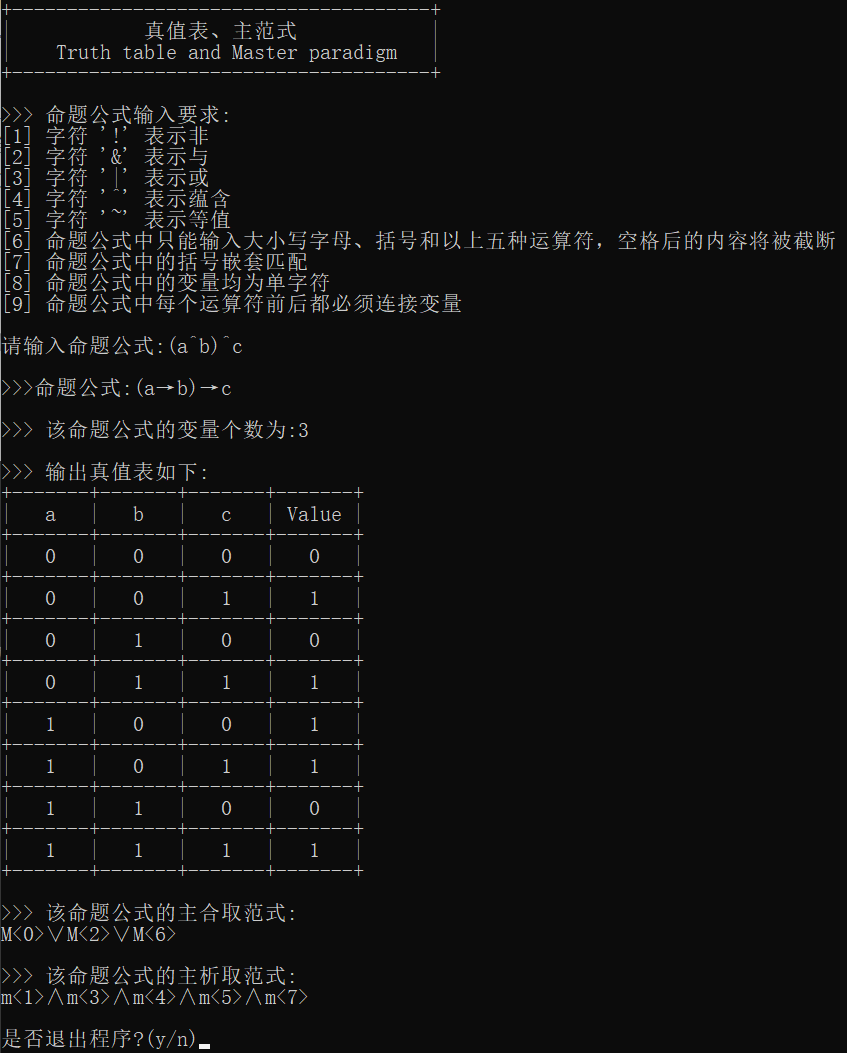


图7.2.4 实验数据4

实验数据分析4：本实验用于测试蕴含运算符的正确性。 输入命题公式为“(a→b)→c”，变量个数为3，真值表输出正确，主合取范 式为 M<0>∧M<2>∧M<6>，主析取范式为 m<1>∨m<3>∨m<4>∨m<5>∨m<7>。

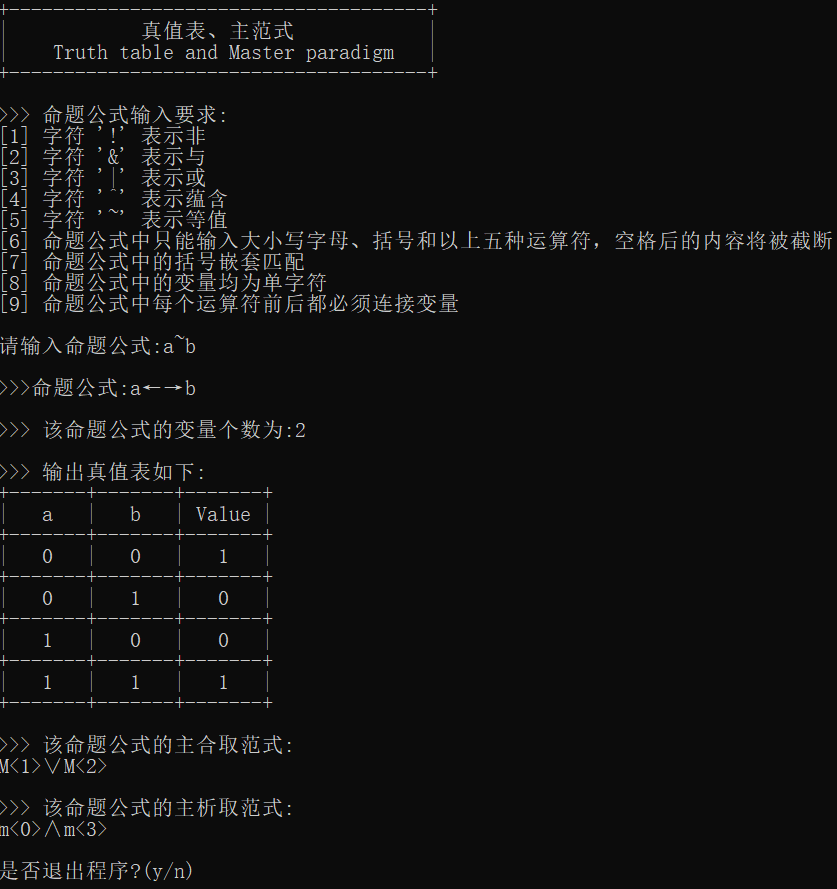


图7.2.5 实验数据5

实验数据分析5：本实验用于测试等值运算符的正确性。 输入命题公式为“a←→b”，变量个数为2，真值表输出正确，主合取范式为 M<1>∧M<2>，成假赋值为 01、10，主析取范式为 m<0>∨m<3>。

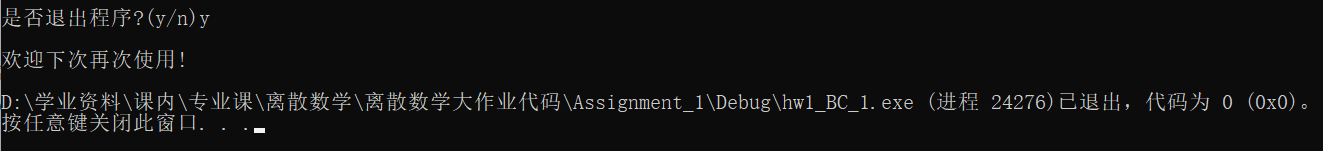


图7.2.6 实验数据6

实验数据分析6：当输入y时，程序退出。

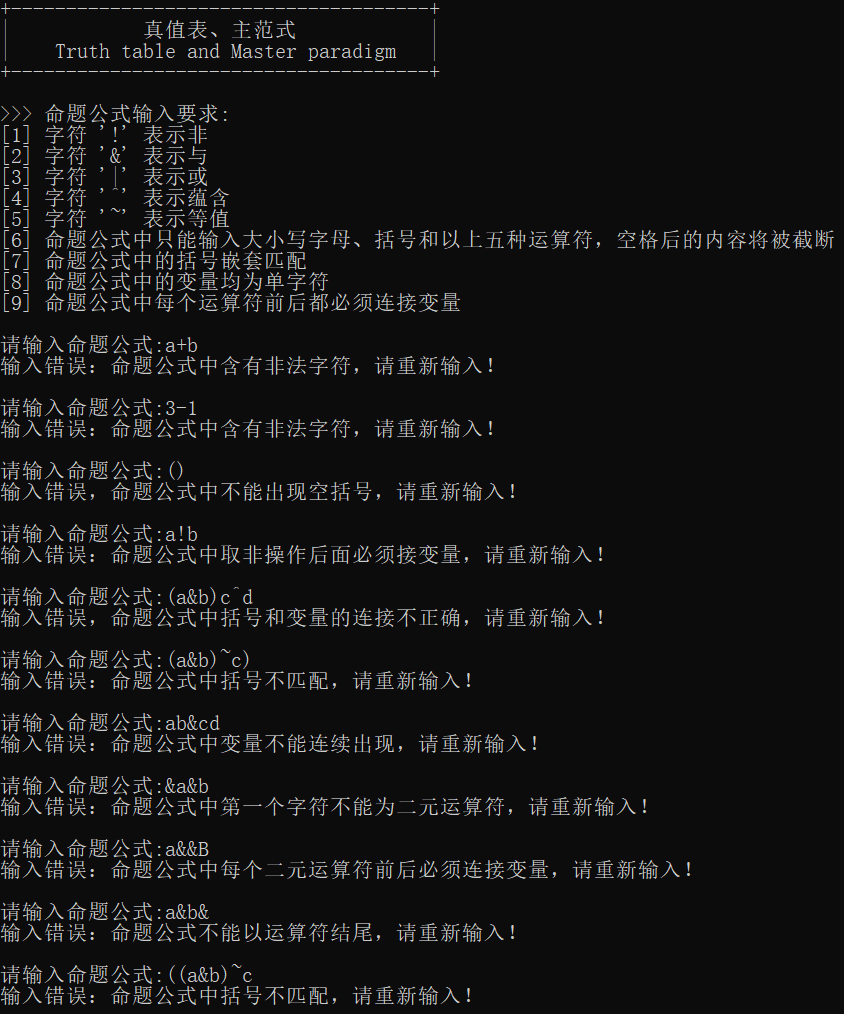


图7.2.7 实验数据7

实验数据分析7：程序会对命题公式的合法性进行检验。当用户输入不合法的命题公式时，程序会打印提示并要求用户重新输入命题公式。程序对命题公式不合法的情况共有11种提示。

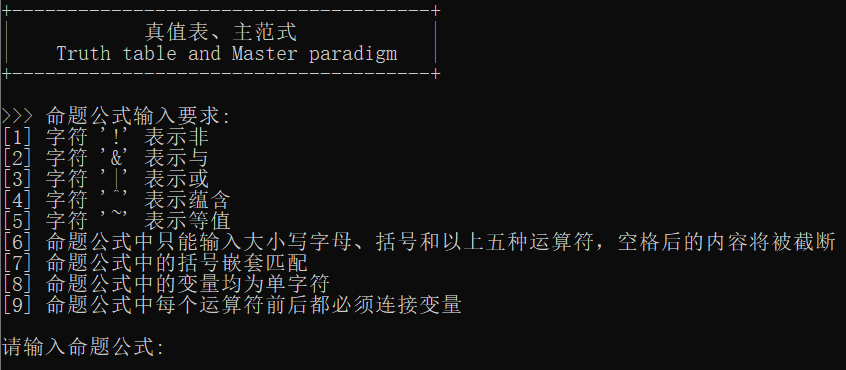


图7.2.8 实验数据8

实验数据分析8：当输入n时，程序清空屏幕并执行下一次真值表与主范式 运算。光标闪烁等待输入命题公式。

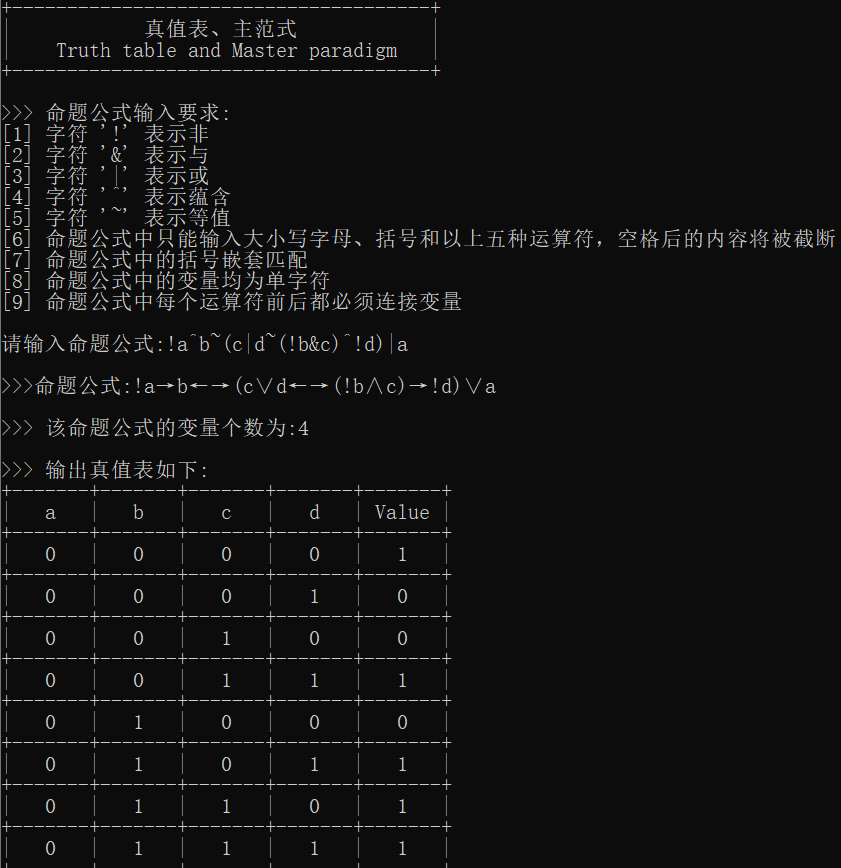


图7.2.9 实验数据9（第一部分）

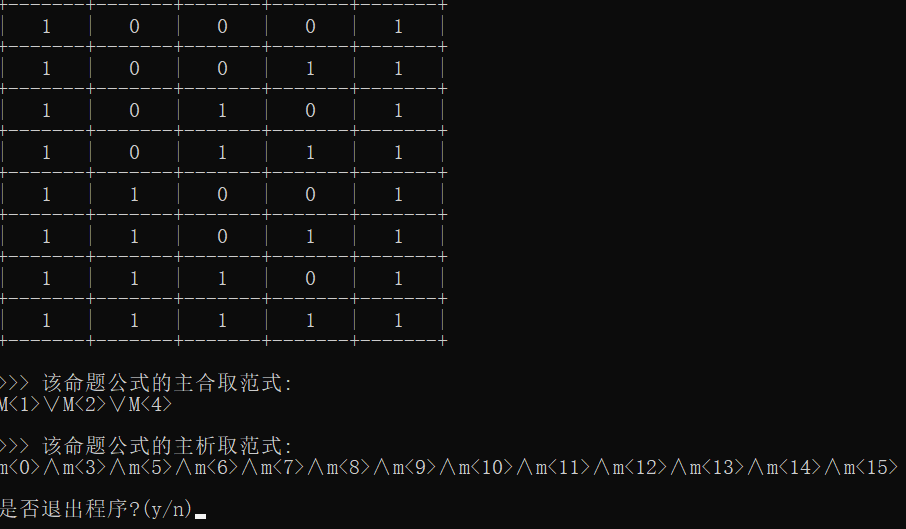


图7.2.10 实验数据9（第二部分）

实验数据分析9：本实验用于测试与、非、或、等值、蕴含运算符的正确性以及括号嵌套匹配的正确性。 输入命题公式为“!a→b←→(c∨d←→(!b∧c)→!d)∨a”，变量个数为 6，真值表输出正确。

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>。

1. **实验心得**

本次实验中，我深入探索了命题逻辑的核心概念，包括联接词、真值表、主析取范式和主合取范式等，并通过实际编程应用这些理论。通过这个过程，我对命题逻辑的理解更加深刻，也提高了我的编程和逻辑分析能力。

实验中，我需要使用基本的逻辑联接词（合取、析取、条件、双向条件）来处理命题变元。这不仅巩固了我对这些概念的理解，还让我了解到如何将它们应用在实际编程中。

真值表是理解和分析逻辑表达式的关键工具。通过编写程序生成真值表，我能够直观地看到不同变元组合下命题逻辑表达式的真值情况。这种视角对于验证逻辑运算的正确性至关重要。

通过这次实验，我学会了如何从真值表中提取信息来构建主析取范式和主合取范式。这不仅提高了我的抽象思维能力，也加深了我对逻辑表达式标准形式的理解。

实验过程中，我使用 C++进行编程，这对于提升我的编程技能非常有帮助。 通过解决实际问题，我更加熟悉了C++语言的特性及其在逻辑运算中的应用。

编写实验报告让我学会了如何总结实验过程和结果，这对于加深理解和记录学习过程非常有帮助。

整个实验不仅是对编程技能的挑战，也是对逻辑思维能力的考验。如何设计算法来解决逻辑问题，需要清晰的逻辑思维，这对我的思维能力是一个很好的锻炼。

1. **程序源文件**
   1. 命题逻辑联接词

#include <iostream>

#include <conio.h>

using namespace std;

/\*

\* Function Name: inputLogicalValue

\* Function: input a logical value from user

\* Input Parameters: char FalseValue = '0'

\* char TrueValue = '1'

\* Return Value: true / false

\*/

bool inputLogicalValue(char FalseValue = '0', char TrueValue = '1')

{

while (true) {

char InputChar = \_getch();

if (InputChar == 0 || InputChar == -32)

InputChar = \_getch();

else if (InputChar == FalseValue || InputChar == TrueValue) {

cout << InputChar <<endl << endl;

return InputChar == TrueValue;

}

}

}

/\*

\* Function Name: main

\* Function: main function

\* Return Value: 0

\*/

int main()

{

do {

system("cls");

cout << "+------------------------------------+" << endl;

cout << "| 命题逻辑连接词 |" << endl;

cout << "| propositional logic connective |" << endl;

cout << "+------------------------------------+" << endl << endl;

cout << "请输入 P 的值 [0/1]: ";

bool p = inputLogicalValue();

cout << "请输入 Q 的值 [0/1]: ";

bool q = inputLogicalValue();

cout << ">>> 合取: P∧Q = " << (p && q) << endl << endl;

cout << ">>> 析取: P∨Q = " << (p || q) << endl << endl;

cout << ">>> 条件: P→Q = " << (!p || q) << endl << endl;

cout << ">>> 双向条件: P←→Q = " << ((!p || q) && (!q || p)) << endl << endl;

cout << "是否退出程序 [y/n]: ";

} while (!inputLogicalValue('n', 'y'));

return 0;

}

* 1. 真值表、主范式

#include <iostream>

#include <string>

#include <map>

#include <conio.h>

#include <stack>

#include <set>

#include <vector>

/\*

\* Function Name: checkOperator

\* Function: check if a character is a logical operator

\* Input Parameters: char c

\* Return Value: true / false

\*/

bool isOperator(char c){

return (c == '!' || c == '&' || c == '|' || c == '~' || c == '^');

}

/\*

\* Function Name: setPriority

\* Function: set the priority of a logical operator

\* Input Parameters: char op

\* Return Value: int

\*/

int setPriority(char op) {

switch (op) {

case ')':

return 5;

case'!':

return 4;

case'&':

case'|':

return 3;

case'^':

return 2;

case'~':

return 1;

case '(':

return 0;

default:

exit(-1);

}

}

/\*

\* Function Name: calculateExpression

\* Function: calculate the value of a expression

\* Input Parameters: char op

\* bool leftValue

\* bool rightValue

\* Return Value: true / false

\*/

bool calculateVariable(char op, bool leftValue, bool rightValue = false) {

switch (op) {

case'!':

return !leftValue;

case'&':

return leftValue && rightValue;

case'|':

return leftValue || rightValue;

case'^':

return !leftValue || rightValue;

case'~':

return leftValue == rightValue;

default:

exit(-1);

}

}

/\*

\* Function Name: replaceLogicalOperators

\* Function: replace logical operators with their corresponding symbols

\* Input Parameters: std::string expression

\* Return Value: std::string

\*/

std::string replaceLogicalOperators(std::string expression){

std::map<char, std::string> OperatorMap = {

{'&', "∧"},

{'|', "∨"},

{'~', "←→"},

{'^', "→"}

};

std::string Result = "";

for (char c : expression)

{

if (c!= '!' && isOperator(c))

Result += OperatorMap[c];

else

Result += c;

}

return Result;

}

/\*

\* Function Name: infixToPostifx

\* Function: convert infix expression to postfix expression

\* Input Parameters: std::string expression

\* Return Value: std::string

\*/

std::string infixToPostifx(std::string expression) {

std::stack<char> operators;

std::string postfixExpression = "";

/\*travel the expression from left to right\*/

for (char c : expression)

{

if (isalpha(c))

postfixExpression += c;

else if (isOperator(c)) {

if (operators.empty())

operators.push(c);

else if (setPriority(c) > setPriority(operators.top()))

operators.push(c);

else {

while (!operators.empty() && setPriority(c) <= setPriority(operators.top())) {

postfixExpression += operators.top();

operators.pop();

}

operators.push(c);

}

}

else if (c == '(')

operators.push(c);

else if (c == ')') {

while (!operators.empty() && operators.top() != '(') {

postfixExpression += operators.top();

operators.pop();

}

if (!operators.empty() && operators.top() == '(') {

operators.pop();

}

}

}

while (!operators.empty()) {

postfixExpression += operators.top();

operators.pop();

}

return postfixExpression;

}

/\*

\* Function Name: calculatePostfixExpression

\* Function: calculate the value of a postfix expression

\* Input Parameters: std::string postfixExpression

\* std::map<char, bool> variableValues

\* Return Value: true / false

\*/

bool calculatePostfixExpression(std::string postfixExpression, std::map<char, bool> variableValues) {

std::stack<bool> values;

for (char c : postfixExpression) {

if(isalpha(c))

values.push(variableValues[c]);

else if (isOperator(c)) {

if (c == '!') {

bool value = values.top();

values.pop();

values.push(!value);

}

else {

bool rightValue = values.top();

values.pop();

bool leftValue = values.top();

values.pop();

values.push(calculateVariable(c, leftValue, rightValue));

}

}

}

return values.top();

}

/\*

\* Function Name: countVariables

\* Function: count the number of variables in a expression

\* Input Parameters: const std::string& expression

\* Return Value: std::set<char>

\*/

std::set<char> countVariables(const std::string expression){

std::set<char> Variables;

for (char c : expression)

{

if (isalpha(c))

Variables.insert(c);

}

return Variables;

}

/\*

\* Function Name: outputTruthTableAndMasterParadigm

\* Function: output the truth table and master paradigm of a expression

\* Input Parameters: const std::string expression

\* Return Value: void

\*/

void outputTruthTableAndMasterParadigm(const std::string expression){

/\*tranform the expression to postfix expression\*/

std::string postfixExpression = infixToPostifx(expression);

/\*count the number of variables\*/

std::set<char> variables = countVariables(expression);

int variableCount = variables.size();

std::cout << ">>> 该命题公式的变量个数为:" << variableCount << std::endl << std::endl;

/\*store master paradigm\*/

std::vector<int> conjunctionParadigm, analyticParadigm;

/\*print the header\*/

std::cout << ">>> 输出真值表如下:" << std::endl;

std::cout << "+";

for (int count = 0; count < variableCount+1; count++)

std::cout << "-------+";

std::cout << std::endl;

std::cout << "|";

for (const auto variable : variables)

std::cout << " " << variable << " |";

std::cout<<" Value |"<< std::endl;

std::cout << "+";

for (int count = 0; count < variableCount + 1; count++)

std::cout << "-------+";

std::cout << std::endl;

/\*print the truth table\*/

std::map<char, bool> variableValues;

for (int i = 0; i < pow(2, variableCount); i++) {

int nthPower = variableCount-1;

std::cout << "|";

/\*set the value of variables\*/

for (const auto variable : variables) {

int temp = static\_cast<int>(pow(2, nthPower));

nthPower--;

variableValues[variable] = i & temp;

std::cout << " " << variableValues[variable] << " |";

}

/\*calculate and output the result\*/

bool value = calculatePostfixExpression(postfixExpression, variableValues);

std::cout << " " << value << " |" << std::endl;

if(value)

analyticParadigm.push\_back(i);

else

conjunctionParadigm.push\_back(i);

std::cout << "+";

for (int count = 0; count < variableCount + 1; count++)

std::cout << "-------+";

std::cout << std::endl;

}

std::cout << std::endl;

std::cout << ">>> 该命题公式的主合取范式:" << std::endl;

for (size\_t count = 0; count < conjunctionParadigm.size(); count++) {

std::cout<<"M<"<<conjunctionParadigm[count]<<">";

if(count!= conjunctionParadigm.size()-1)

std::cout<<"∨";

}

std::cout << std::endl << std::endl;

std::cout << ">>> 该命题公式的主析取范式:" << std::endl;

for (size\_t count = 0; count < analyticParadigm.size(); count++) {

std::cout<<"m<"<<analyticParadigm[count]<<">";

if(count!= analyticParadigm.size()-1)

std::cout<<"∧";

}

std::cout << std::endl << std::endl;

}

/\*

\* Function Name: isValidExpression

\* Function: Check if a expression is valid

\* Input Parameters: const std::string& expression

\* Return Value: true / false

\*/

bool checkExpression(const std::string &expression){

if (expression.empty()){

std::cout << "输入错误：命题公式为空，请重新输入！" << std::endl << std::endl;

return false;

}

std::stack<char> parentheses;

char prev\_char = '\0';

/\*travelling the expression from left to right\*/

for (char c : expression)

{

if (!isalpha(c) && (!isOperator(c)) && c != '(' && c != ')')

{

std::cout << "输入错误：命题公式中含有非法字符，请重新输入！" << std::endl << std::endl;

return false;

}

if (prev\_char == '\0' && (c == '&' || c == '|' || c == '~' || c == '^'))

{

std::cout << "输入错误：命题公式中第一个字符不能为二元运算符，请重新输入！" << std::endl << std::endl;

return false;

}

if (prev\_char == '(' && c == ')')

{

std::cout<<"输入错误，命题公式中不能出现空括号，请重新输入！" << std::endl << std::endl;

return false;

}

if (c == '!' && prev\_char == '!')

{

std::cout << "输入错误：命题公式中不能连续出现取非操作，请重新输入！" << std::endl << std::endl;

return false;

}

if (c == '!' && isalpha(prev\_char))

{

std::cout << "输入错误：命题公式中取非操作后面必须接变量，请重新输入！" << std::endl << std::endl;

return false;

}

if ((prev\_char == ')' && isalpha(c)) || c == '(' && isalpha(prev\_char))

{

std::cout<<"输入错误，命题公式中括号和变量的连接不正确，请重新输入！" << std::endl << std::endl;

return false;

}

if (c == '(')

parentheses.push(c);

else if (c == ')')

{

if (parentheses.empty())

{

std::cout << "输入错误：命题公式中括号不匹配，请重新输入！" << std::endl << std::endl;

return false;

}

parentheses.pop();

}

if (isalpha(c) && isalpha(prev\_char))

{

std::cout << "输入错误：命题公式中变量不能连续出现，请重新输入！" << std::endl << std::endl;

return false;

}

if ((c == '&' || c == '|' || c == '~' || c == '^') && (!isalpha(prev\_char) && prev\_char!= ')')) {

std::cout << "输入错误：命题公式中每个二元运算符前后必须连接变量，请重新输入！" << std::endl << std::endl;

return false;

}

prev\_char = c;

}

if (isOperator(prev\_char))

{

std::cout << "输入错误：命题公式不能以运算符结尾，请重新输入！" << std::endl << std::endl;

return false;

}

if (!parentheses.empty())

{

std::cout << "输入错误：命题公式中括号不匹配，请重新输入！" << std::endl << std::endl;

return false;

}

return true;

}

/\*

\* Function Name: inputLogicalValue

\* Function: input a logical value from user

\* Input Parameters: char FalseValue = '0'

\* char TrueValue = '1'

\* Return Value: true / false

\*/

bool inputLogicalValue(char FalseValue = '0', char TrueValue = '1'){

while (true) {

char InputChar = \_getch();

if (InputChar == 0 || InputChar == -32)

InputChar = \_getch();

else if (InputChar == FalseValue || InputChar == TrueValue) {

std::cout << InputChar << std::endl << std::endl;

return InputChar == TrueValue;

}

}

}

/\*

\* Function Name: main

\* Function: main function

\* Return Value: 0

\*/

int main() {

do {

system("cls");

/\*print the title\*/

std::cout << "+--------------------------------------+" << std::endl;

std::cout << "| 真值表、主范式 |" << std::endl;

std::cout << "| Truth table and Master paradigm |" << std::endl;

std::cout << "+--------------------------------------+" << std::endl << std::endl;

/\*output the requirement of input\*/

std::cout << ">>> 命题公式输入要求:" << std::endl;

std::cout << "[1] 字符 '!' 表示非" << std::endl;

std::cout << "[2] 字符 '&' 表示与" << std::endl;

std::cout << "[3] 字符 '|' 表示或" << std::endl;

std::cout << "[4] 字符 '^' 表示蕴含" << std::endl;

std::cout << "[5] 字符 '~' 表示等值" << std::endl;

std::cout << "[6] 命题公式中只能输入大小写字母、括号和以上五种运算符，空格后的内容将被截断" << std::endl;

std::cout << "[7] 命题公式中的括号嵌套匹配" << std::endl;

std::cout << "[8] 命题公式中的变量均为单字符" << std::endl;

std::cout << "[9] 命题公式中每个运算符前后都必须连接变量" << std::endl<< std::endl;

/\*input the formula\*/

std::string Formula;

do {

std::cout << "请输入命题公式:" ;

std::cin >> Formula;

} while (!checkExpression(Formula));

std::cout << std::endl;

/\*output the formula and truth table\*/

std::cout << ">>>命题公式:" <<replaceLogicalOperators(Formula) << std::endl<< std::endl;

outputTruthTableAndMasterParadigm(Formula);

std::cout << "是否退出程序?(y/n)" ;

} while (!inputLogicalValue('n', 'y'));

std::cout << "欢迎下次再次使用!" << std::endl;

return 0;

}