同濟大學
TONGJI UNIVERSITY

《离散数学》课程实验报告

题目 命题逻辑联接词、真值表、主范式

姓　名 林继申

学　号 2250758

学　院 软件学院

专　业 软件工程

教　师 李　冰

二〇二三 年 十二 月 二十二 日

目录

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。

2 实验内容

2.1 命题逻辑联接词

在这部分实验中,目标是熟悉基本的命题逻辑联接词:合取(∧)、析取(∨)、条件（→）和双向条件（↔)。程序将从用户那里接收两个命题变元 P 和 Q 的真值，并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。

2.2 真值表、主范式

这部分实验的目标是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)。用户输入一个命题逻辑公式，程序首先验证该公式的有效性，然后生成并显示其真值表。基于真值表，程序计算并显示该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。

通过真值表，可以确定哪些变量组合使公式成真或成假，进而构造出相应的极小项或极大项，最后将这些项以逻辑“或”或逻辑“与”的形式组合起来，形

成 DNF 或 CNF。

3 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

4 实验原理

4.1 合取

合取是一种二元逻辑运算。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。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 4.1.1 合取运算真值表

P	Q	P∧Q
1	1	1
1	0	0
0	1	0
0	0	0

4.2 析取

析取是一种二元逻辑运算。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。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 4.2.1 析取运算真值表

P	Q	P∨Q
1	1	1
1	0	1
0	1	1
0	0	0

4.3 条件

条件是一种二元逻辑运算，也称为蕴含。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。如果第一个命题（前件）为真，则第二个命题（后件）也必须为真，整个条件才为真。唯一使条件为假的情况是前件为真而后件为假。

表 4.3.1 条件运算真值表

P	Q	P→Q
1	1	1
1	0	0
0	1	1
0	0	1

4.4 双向条件

双向条件是一种二元逻辑运算，也称为等价。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。当两个命题具有相同的真值时（要么都为真，要么都为假），它们的双向条件为真。

表 4.4.1 双向条件运算真值表

P	Q	P↔Q
1	1	1
1	0	0
0	1	0
0	0	1

4.5 真值表

真值表是一个表格，用于表示逻辑运算符在其操作数（通常是命题变量）的所有可能组合下的结果。表中的每一行代表了一种组合，列出了在这种组合下每个操作数和结果的真值。真值表通常由行和列组成。行代表不同的输入组合，而列代表这些输入和输出的真值。最左侧的列是输入变量，最右侧的列是这些输入变量经过逻辑运算后的输出。

真值表被广泛用于分析和理解复杂的逻辑表达式。它有助于确定表达式在不同的真值输入下的结果。真值表可以用来证明两个逻辑表达式是否逻辑等价，即在所有可能的输入情况下是否总是给出相同的结果。

4.6 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。

4.7 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。

5 实验过程（命题逻辑联接词）

5.1 实验思路

在这个实验中，目的是设计一个程序来演示命题逻辑联接词的功能。实验通过输入两个命题变元的真值，然后计算并展示出这些变元经过合取（∧）、析取（∨）、条件（→）和双向条件（↔）等联接词处理后的结果。

5.2 实验设计

5.2.1 数据结构设计

由于这个程序主要涉及基本的逻辑运算，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。

5.2.2 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。

5.2.3 程序主体架构设计

程序以一个主函数 main 作为入口，其中包含整个逻辑流程。程序首先清理屏幕，然后打印出界面和说明。接着，通过 inputLogicalValue 函数获取用户输入的命题变元 P 和 Q 的真值。之后，程序计算并显示这些变元经过不同联接词处理的结果。最后，程序询问用户是否退出。

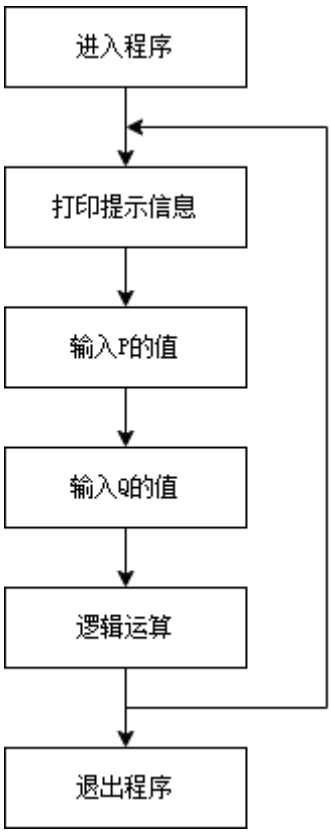


图 5.2.3.1 程序主体架构设计流程图

5.3 程序功能实现

5.3.1 逻辑值输入功能的实现

inputLogicalValue 函数用于输入逻辑值，专门用于处理二进制（真/假）值的用户输入，在这里分别用'0'和'1'或者通过参数 falseValue 和 trueValue 指定的其他字符来表示。

程序功能实现思路：

(1)falseValue 表示假的值，默认为'0'。trueValue 表示真的值，默认为'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（如方向键），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。

5.3.2 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。

5.4 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。

6 实验过程（真值表、主范式）

6.1 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）。实验的重点在于有效解析命题逻辑表达式，生成真值表，并从中导出 DNF 和 CNF。

6.2 实验设计

6.2.1 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。

`std::queue<char>`：存储转换后的后缀表达式。

6.2.2 结构体与类设计

本实验未使用结构体或类设计，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。

6.2.3 程序主体架构设计

程序的主体架构围绕 `main` 函数展开，包括用户界面的呈现、命题逻辑表达式的输入、有效性检查、真值表的生成以及 DNF 和 CNF 的导出。

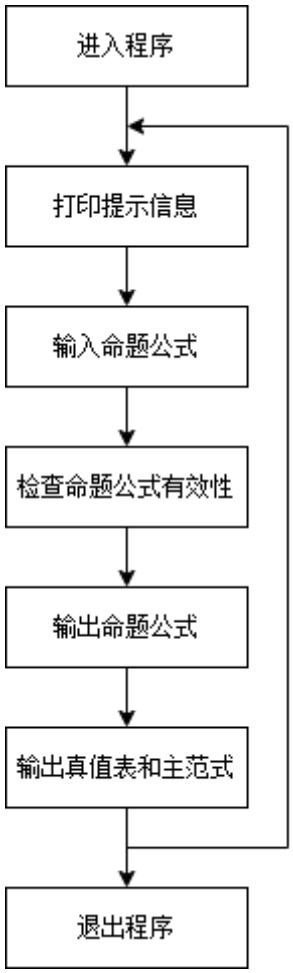


图 6.2.3.1 程序主体架构设计流程图

6.3 程序功能实现

6.3.1 命题公式合法性检验功能

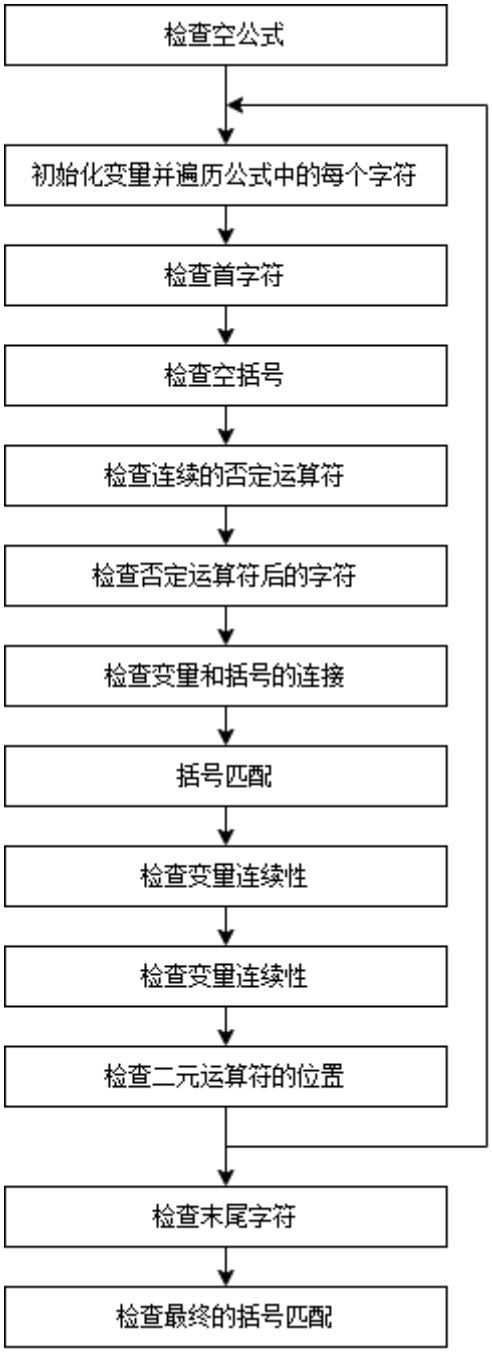


图 6.3.1.1 命题公式合法性检验功能实现流程图

`isValidExpression` 函数检查多种语法规则，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（Negation）

②字符'&'表示与（Conjunction）

③字符'|'表示或（Disjunction）

④字符'^'表示蕴含（Implication）

⑤字符'~'表示等值（Equivalence）

⑥命题公式中只存在以下 59 种字符: a-z A-Z ！ & | ^ ~ ( )

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（"!!a"请输入为"!(!a)"）

程序功能实现思路：

(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 charprevious 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 isOperator 函数检查）或括号。如果不是，提示“命题公式存在非法字符输入，请重新输入!”并返回 false；

(4)检查首字符：如果公式的首字符是二元运算符（'&','|','~','^'），则提示“命题公式不能以二元运算符开始，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“命题公式存在空括号，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“命题公式存在不合法的连续取非操作，请重新输入!”并返回 false；

(7)检查否定运算符后的字符：如果'!'后紧跟字母，提示“命题公式中取非运算符前不可连接变量，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“命题公式括号不匹配，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“命题公式仅适用于单字符变量，不适用于多字符变量，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“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“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“命题公式括号不匹配，请重新输入!”并返回 false。

6.3.2 命题公式输出功能

replaceSymbols 函数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。

6.3.3 命题公式变量提取功能

extractVariables 函数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。

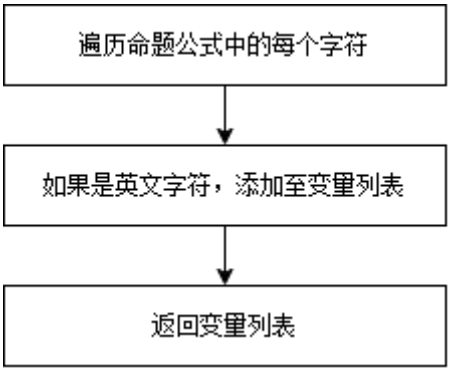


图 6.3.3.1 命题公式变量提取功能实现流程图

6.3.4 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 extractVariables 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：使用 printSeparator 函数打印表格的头部分隔线。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 numVariables 个变量，存在 2^numVariables 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，通过位操作设置每个变量的值（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：中缀到后缀的转换：使用 infixToPostfix 函数将命题公式从中缀表达式转换为后缀表达式，以便于计算。在 infixToPostfix 函数中创建一个字符堆栈 ops 来存储运算符和括号。遍历表达式的每个字符，对于每个运算符，弹出堆栈中所有优先级高于或等于当前运算符的运算符，并将它们推送到输出队列。遇到括号时，特殊处理以保持正确的嵌套；

(5)后缀表达式计算：调用 calculatePostfix 函数，使用刚才设置的变量赋值来计算后缀表达式的结果。在 calculatePostfix 函数中，使用布尔值堆栈 s。遍历后缀表达式的每个字符，对于变量，将其值（从映射 values 中获取）压入堆栈。对于运算符，从堆栈中弹出所需数量的操作数，应用运算符，并将结果压回堆栈；

(6)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存储在一个向量中。在所有赋值组合遍历完成后，打印每个赋值组合下公式的计算结果。

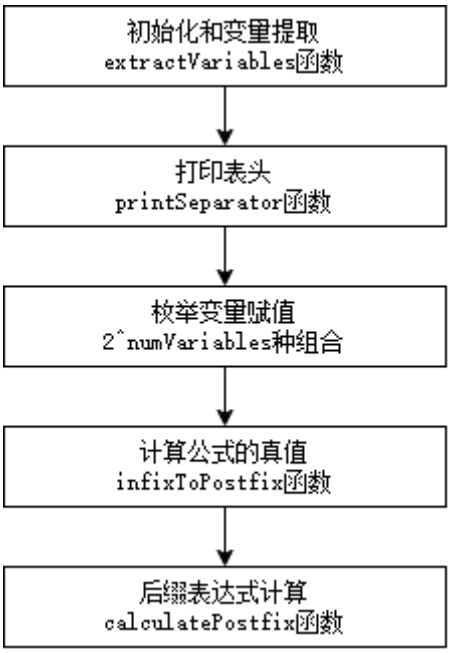


图 6.3.4.1 真值表输出功能实现流程图

6.3.5 主合取范式与成假赋值输出功能

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。成假赋值则是指使得整个公式为假的变量赋值。

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 results 向量中。该向量存储了对应于每种变量赋值的公式真值（真或假）；

(2)判断公式是否恒真：通过检查 results 向量中是否所有元素都为真（即没有假值），来判断公式是否是恒真的。如果是，CNF 直接输出为“1”；

(3)输出主合取范式：如果公式不是恒真，遍历 results 向量。对于每个使公式为假的赋值（在 results 中为 0 的项），将其转换成相应的 CNF 子句，并加入到 CNF 表达式中。使用“M<编号>”来代表每个使公式为假的赋值组合，其中“编号”是变量赋值的二进制表示对应的十进制数字；

(4)输出成假赋值：对于每个使公式为假的赋值组合，使用 printBinary 函数打印出相应的变量赋值情况。这些赋值是以二进制格式显示的，代表了哪些变量组合会使公式的结果为假。

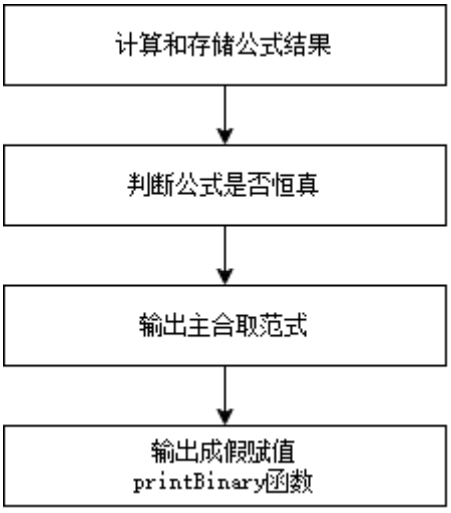


图 6.3.5.1 真值表输出功能实现流程图

6.3.6 主析取范式与成真赋值输出功能

主析取范式与成真赋值的输出功能与主合取范式与成假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。成真赋值则是指使得整个公式为真的变量赋值。

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 results 向量中。该向量存储了对应于每种变量赋

值的公式真值（真或假）；

(2)判断公式是否恒假：通过检查 results 向量中是否所有元素都为假（即没有真值），来判断公式是否是恒假的。如果是，DNF 直接输出为“0”；

(3)输出主析取范式：如果公式不是恒假，遍历 results 向量。对于每个使公式为真的赋值（在 results 中为 1 的项），将其转换成相应的 DNF 子句，并加入到 DNF 表达式中。使用“m<编号>”来代表每个使公式为真的赋值组合，其中“编号”是变量赋值的二进制表示对应的十进制数字；

(4)输出成真赋值：对于每个使公式为真的赋值组合，使用 printBinary 函数打印出相应的变量赋值情况。这些赋值是以二进制格式显示的，代表了哪些变量组合会使公式的结果为真。

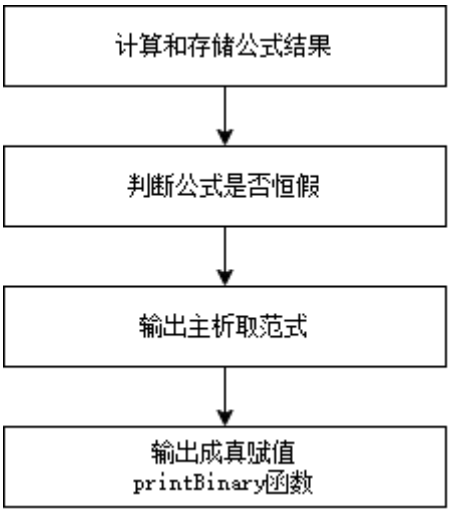


图 6.3.6.1 真值表输出功能实现流程图

6.3.7 退出程序功能的实现

同 5.3.2 退出程序功能的实现。

6.4 核心算法实现

6.4.1 真值表计算算法

真值表计算算法是逻辑表达式求解的基础。首先，extractVariables 函数从表达式中提取唯一变量集合，确定涉及的逻辑变量数量。然后，infixToPostfix 函数将表达式从中缀形式转换为后缀形式（逆波兰表示法），这是为了便于接下来的计算过程。在后缀表达式计算中，calculatePostfix 函数利用栈来处理后缀表达式中的运算符和操作数，计算出每个变量赋值组合下表达式的布尔值。最后，outputTruthTableAndNormalForms 函数通过遍历所有可能的变量赋值组合（使用位操作）来构建完整的真值表，记录表达式在各种不

同变量赋值下的真或假结果。

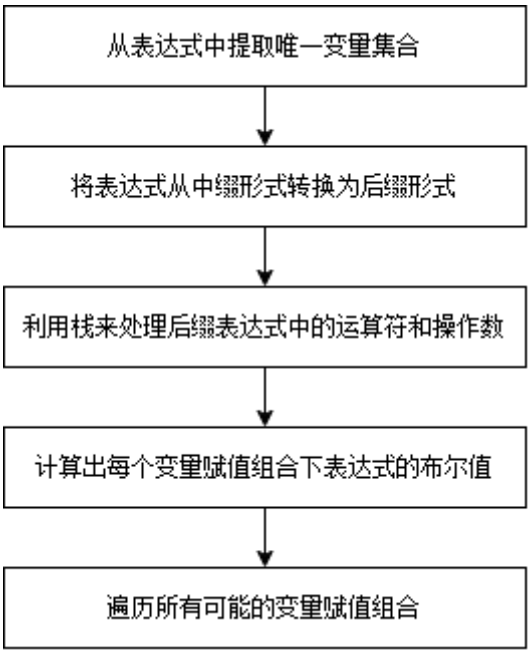


图 6.4.1.1 真值表计算算法实现流程图

6.4.2 主合取范式与成假赋值计算算法

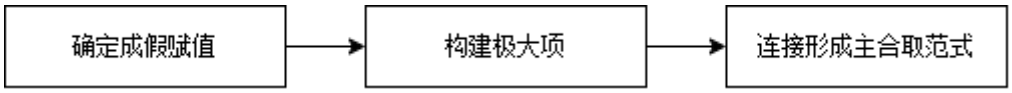


图 6.4.2.1 主合取范式与成假赋值计算算法实现流程图

主合取范式（CNF）与成假赋值的计算是基于真值表的结果进行的。这个算法通过检查真值表中哪些行使表达式结果为假，从而确定成假的变量赋值。对于每个使表达式成假的变量赋值，算法构建一个子句，这些子句在逻辑上用“与”连接起来，形成主合取范式。如果表达式在所有变量赋值下总是为真，那么 CNF 简化为逻辑常数 1。

6.4.3 主析取范式与成真赋值计算算法

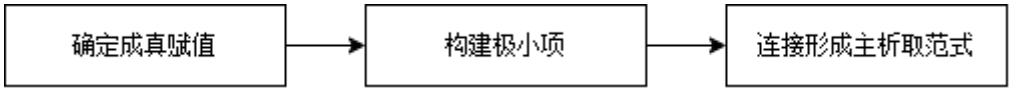


图 6.4.3.1 主析取范式与成真赋值计算算法实现流程图

主析取范式（DNF）与成真赋值的计算过程与 CNF 的计算类似。该算法检查真值表中结果为真的每一行，并对每个使表达式成真的变量赋值构建一个子句。这些子句在逻辑上用“或”连接起来，形成主析取范式。如果表达式在所有变量赋值下总是为假，则 DNF 简化为逻辑常数 0。

7 实验数据分析

7.1 命题逻辑联接词

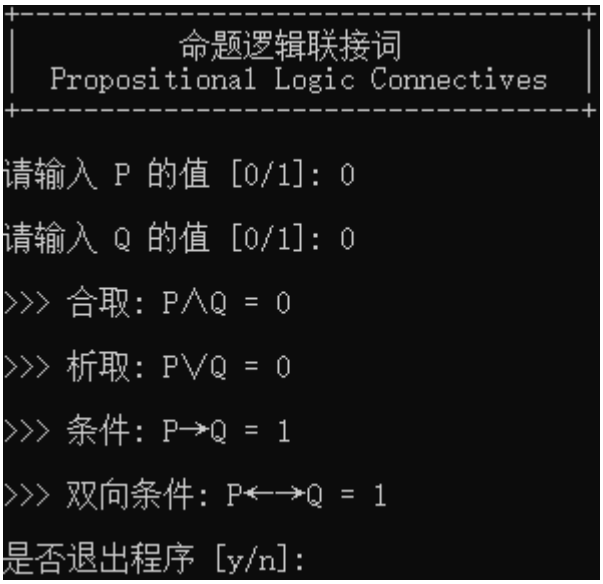


图 7.1.1 实验数据 1

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。

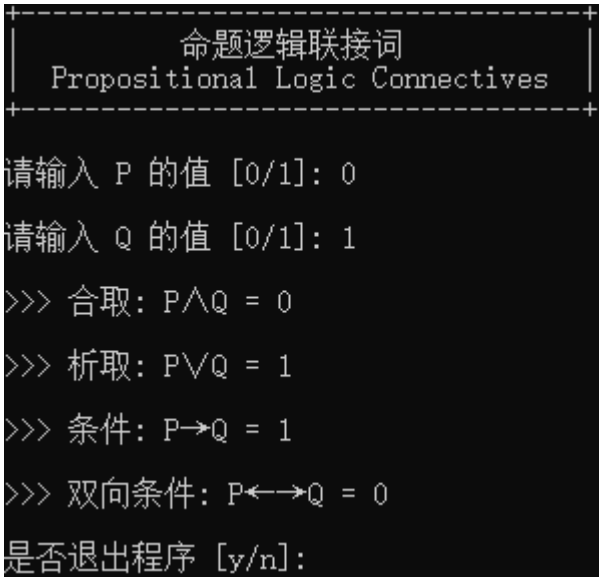


图 7.1.2 实验数据 2

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。

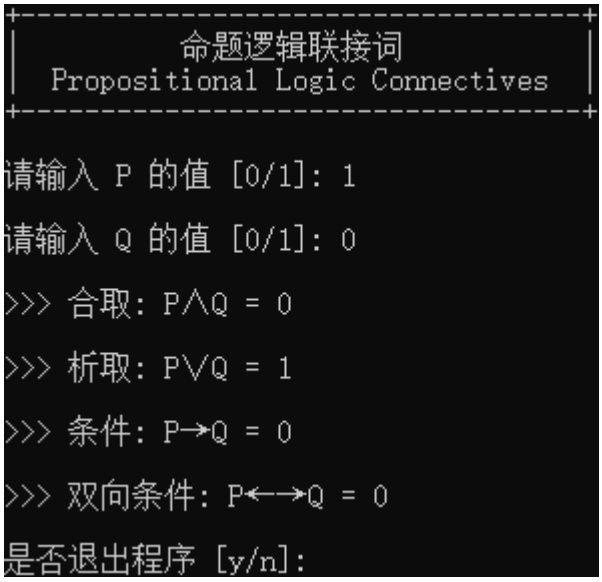


图 7.1.3 实验数据 3

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。

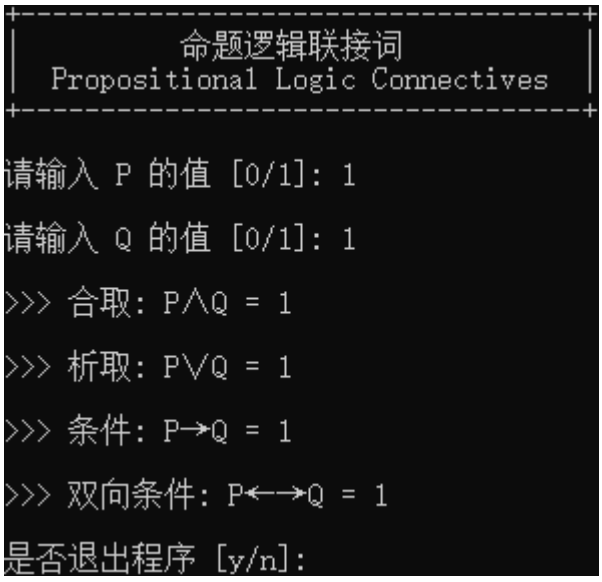


图 7.1.4 实验数据 4

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。

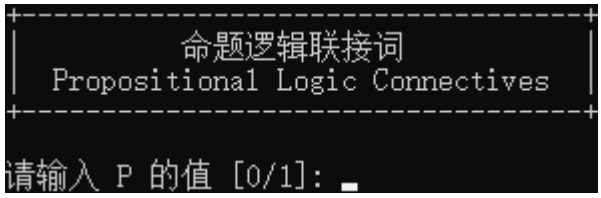


图 7.1.5 实验数据 5

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。

7.2 真值表、主范式

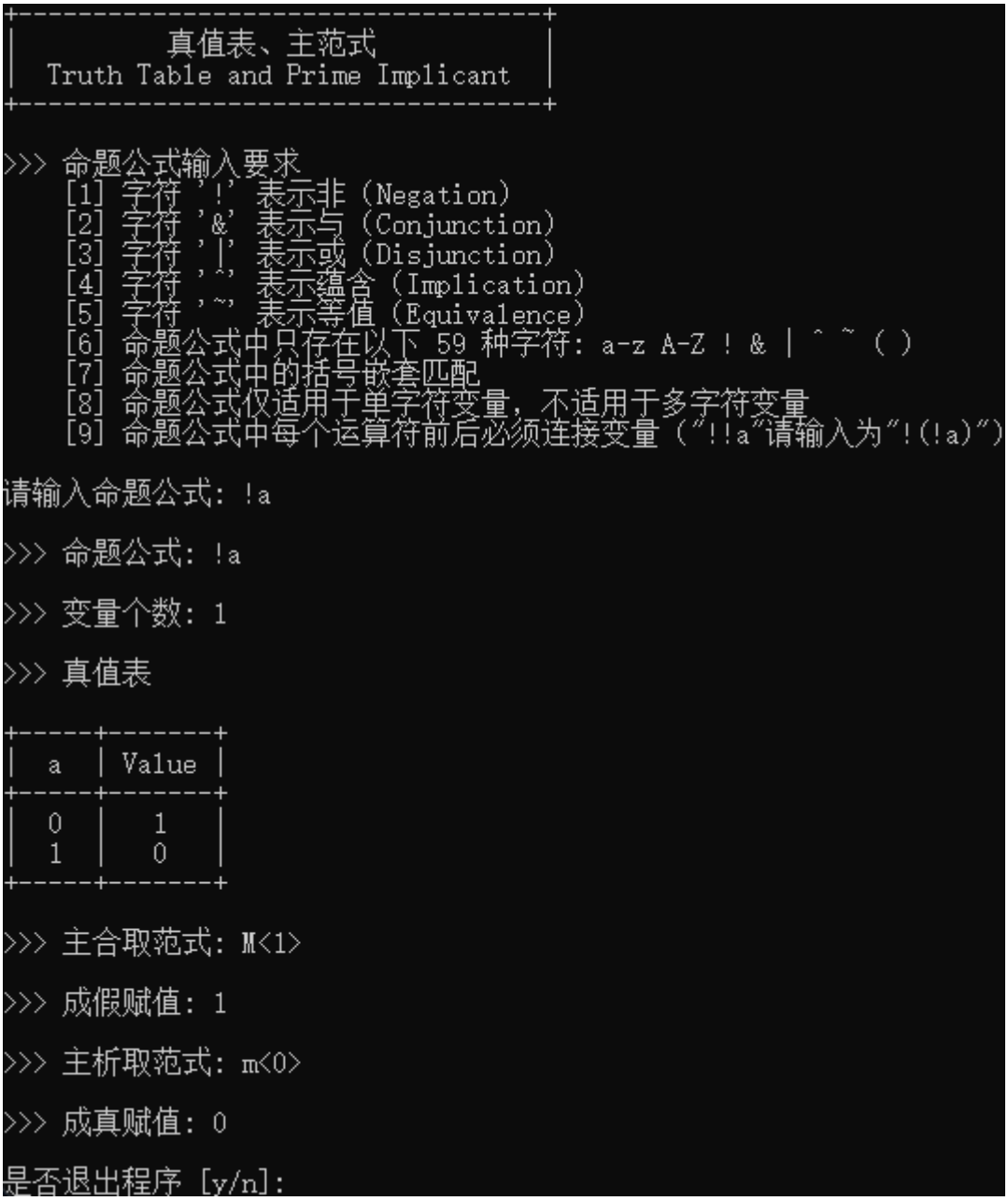


图 7.2.1 实验数据 1

实验数据分析 1：本实验用于测试非（Negation）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>,

成假赋值为 1，主析取范式为 m<0>，成真赋值为 0。

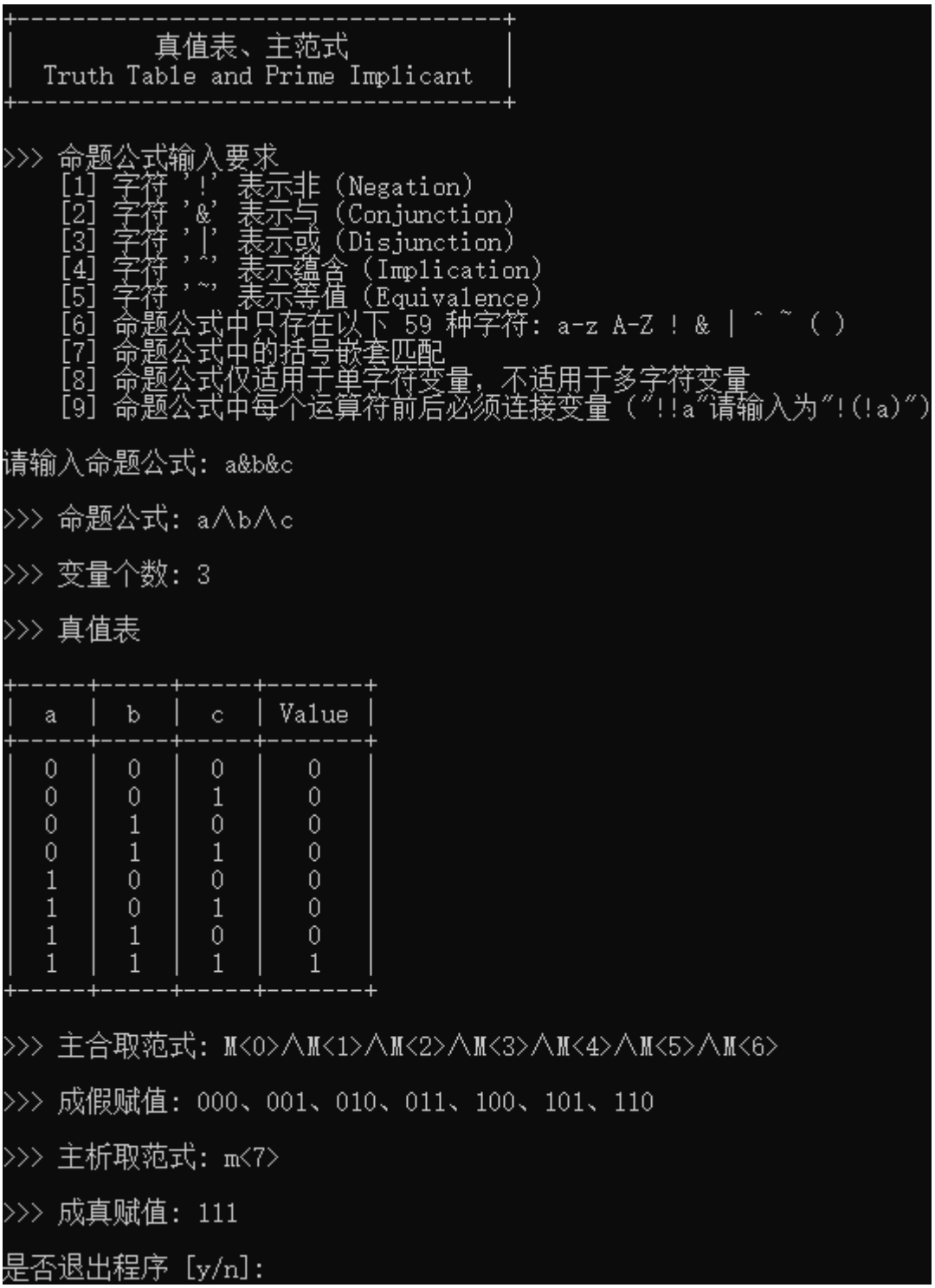


图 7.2.2 实验数据 2

实验数据分析 2：本实验用于测试与（Conjunction）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>，成假赋值为 000、001、010、011、100、101、110，主析取范式为 m<7>，成真赋值为 111。

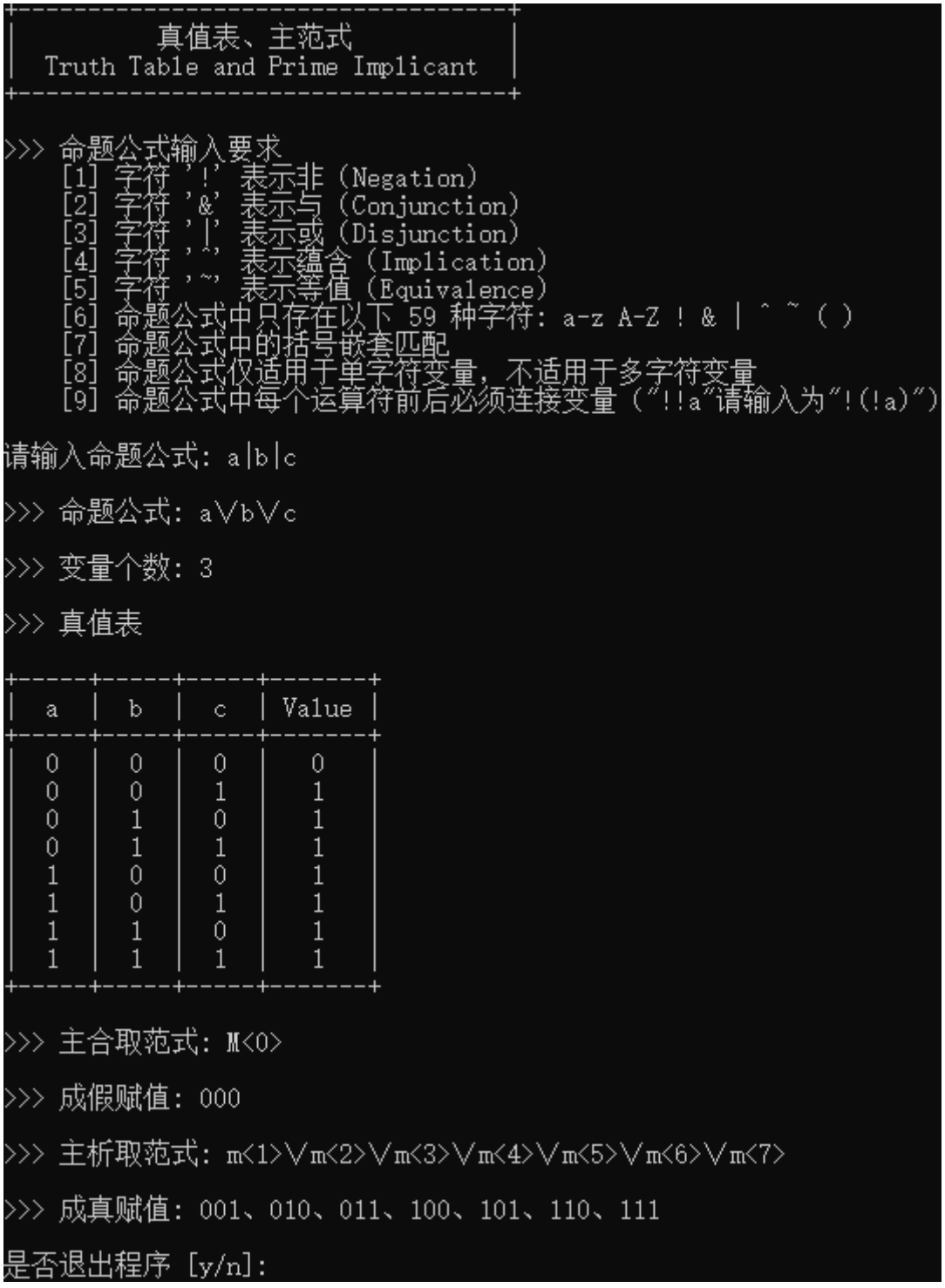


图 7.2.3 实验数据 3

实验数据分析 3：本实验用于测试或（Disjunction）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>，成真赋值为 001、010、011、100、101、110、111。

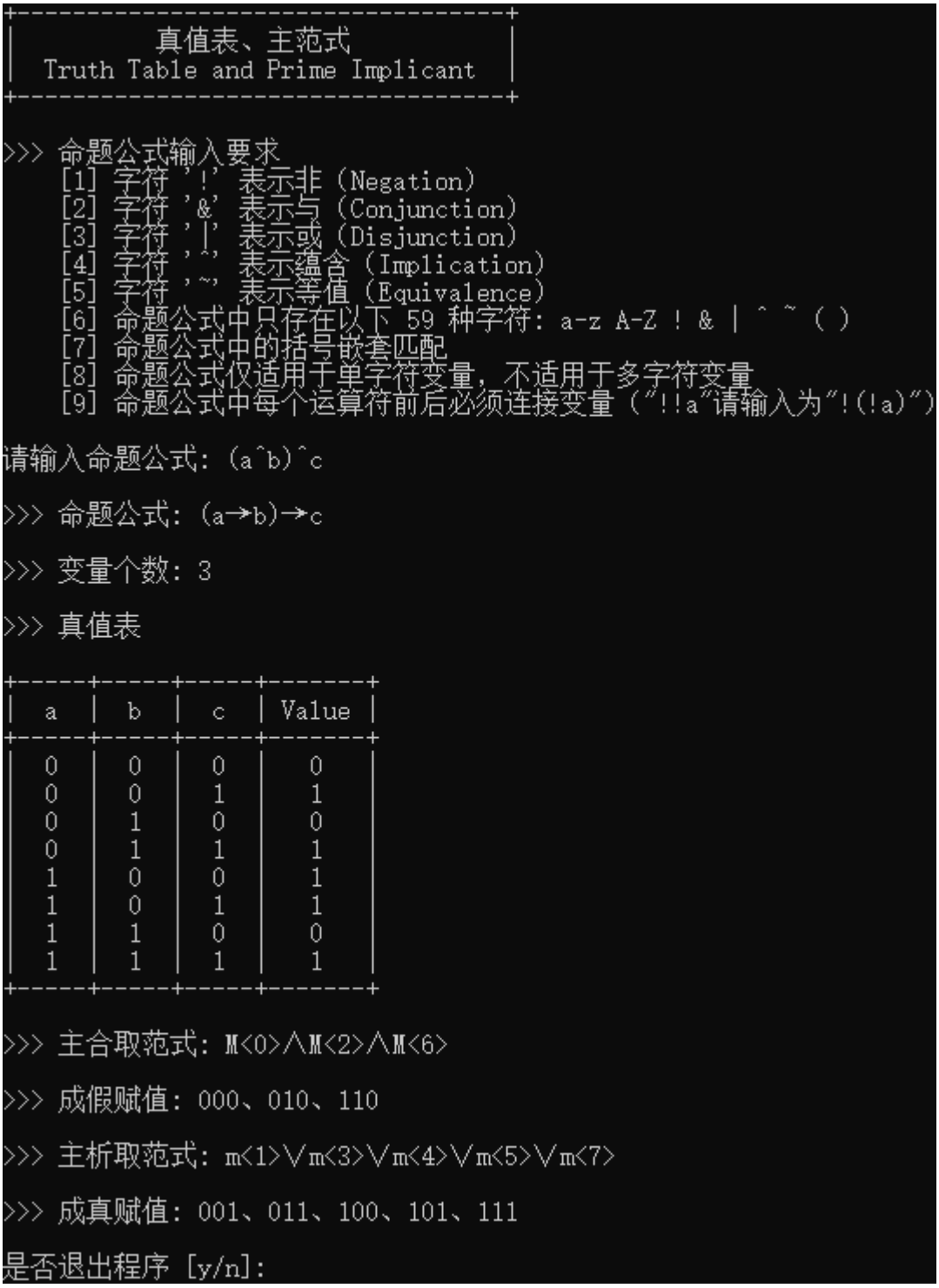


图 7.2.4 实验数据 4

实验数据分析 4：本实验用于测试蕴含（Implication）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>，成假赋值为 000、010、110，主析取范式为 m<1>∨m<3>∨m<4>∨m<5>∨m<7>，成真赋值为 001、011、100、101、111。

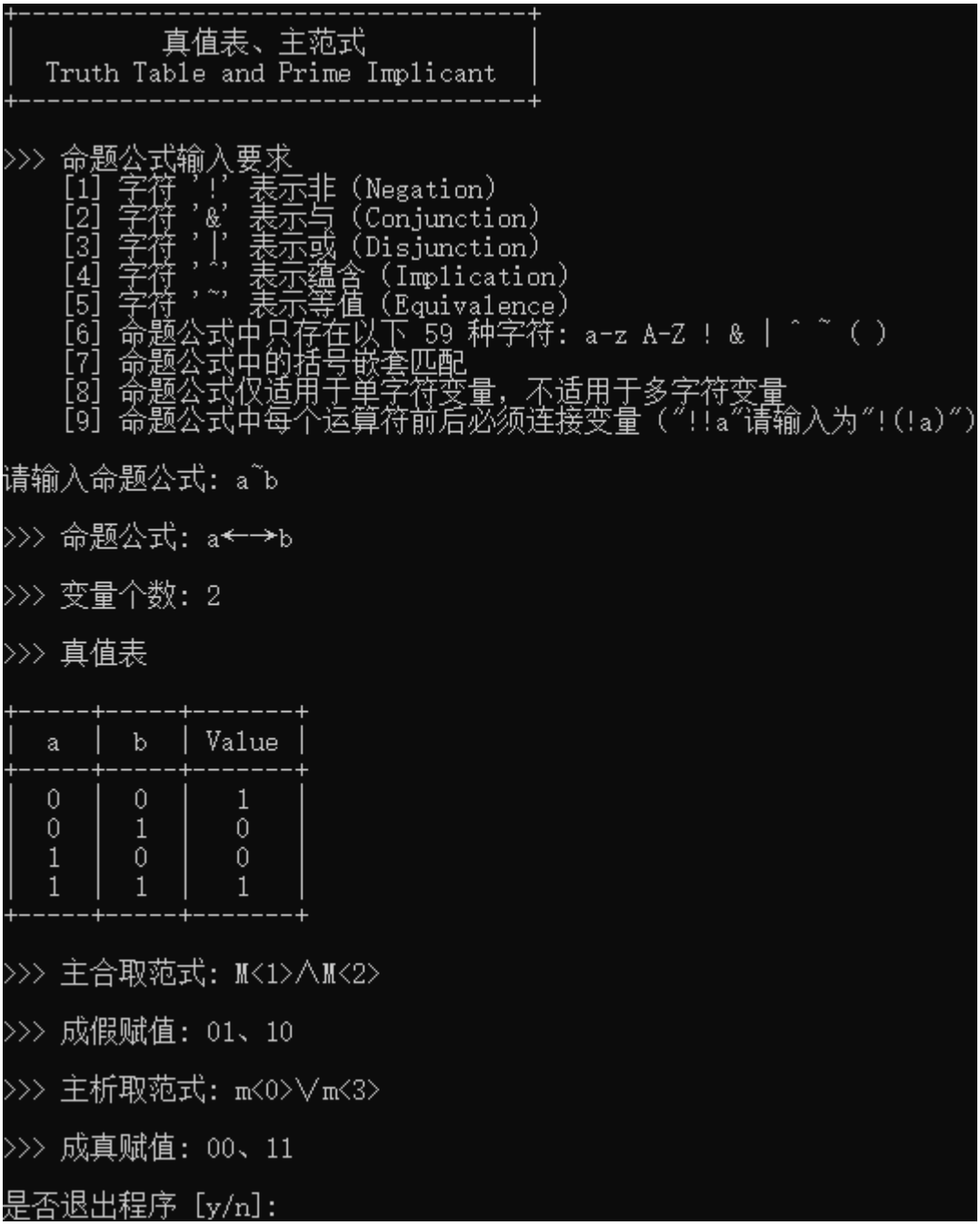


图 7.2.5 实验数据 5

实验数据分析 5：本实验用于测试等值（Equivalence）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>，成真赋值为 00、11。

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。

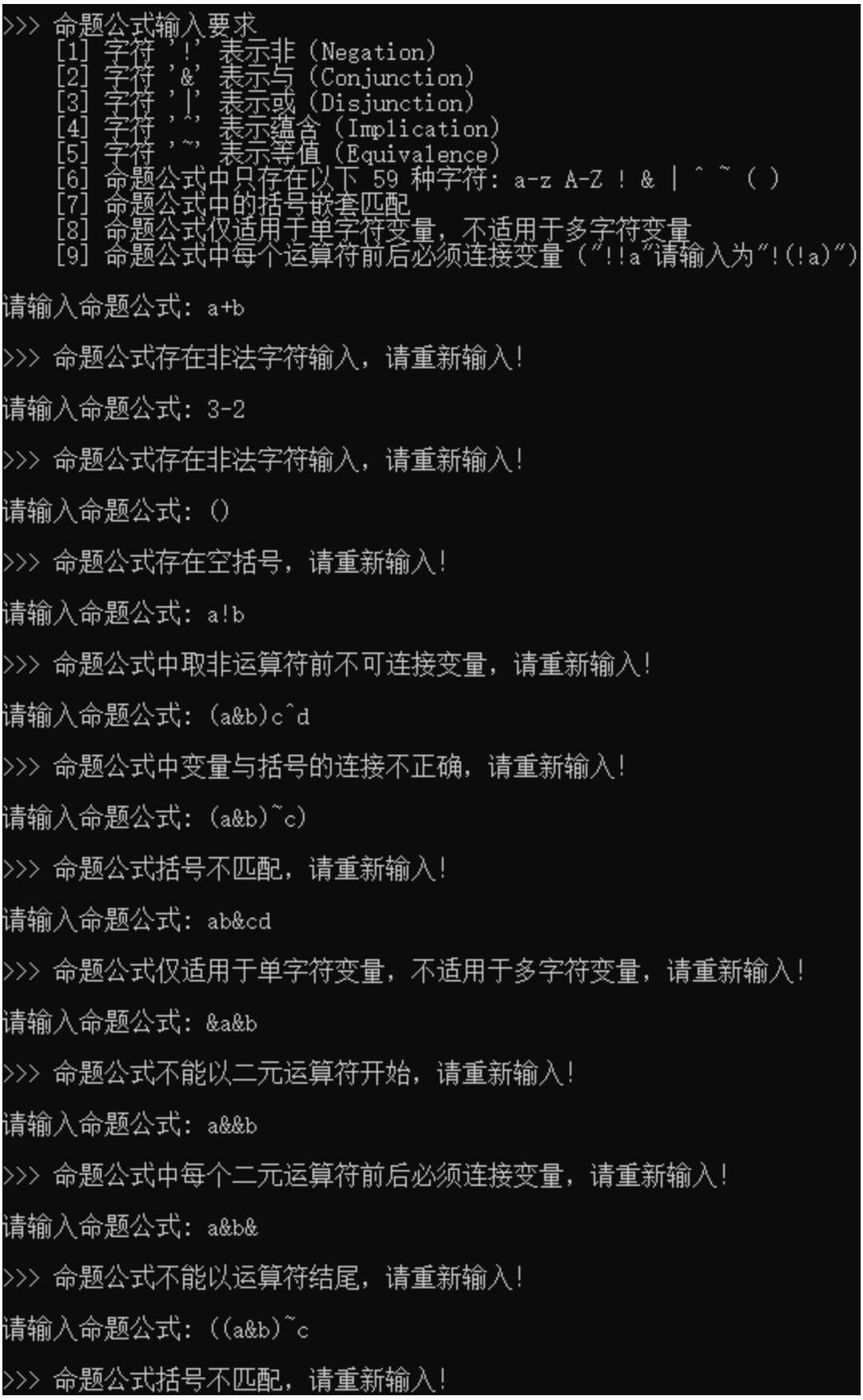


图 7.2.7 实验数据 7

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。

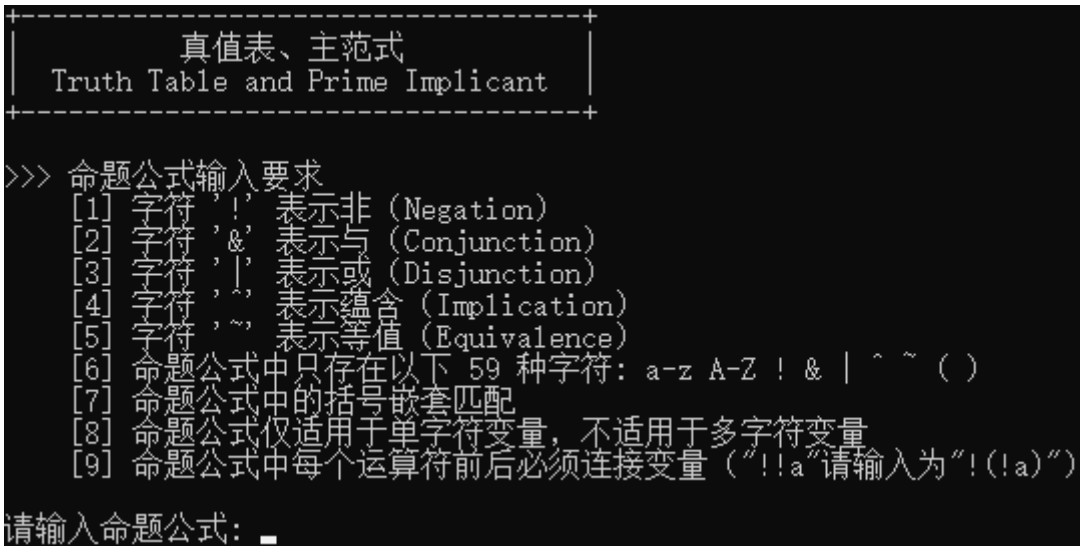


图 7.2.8 实验数据 8

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。

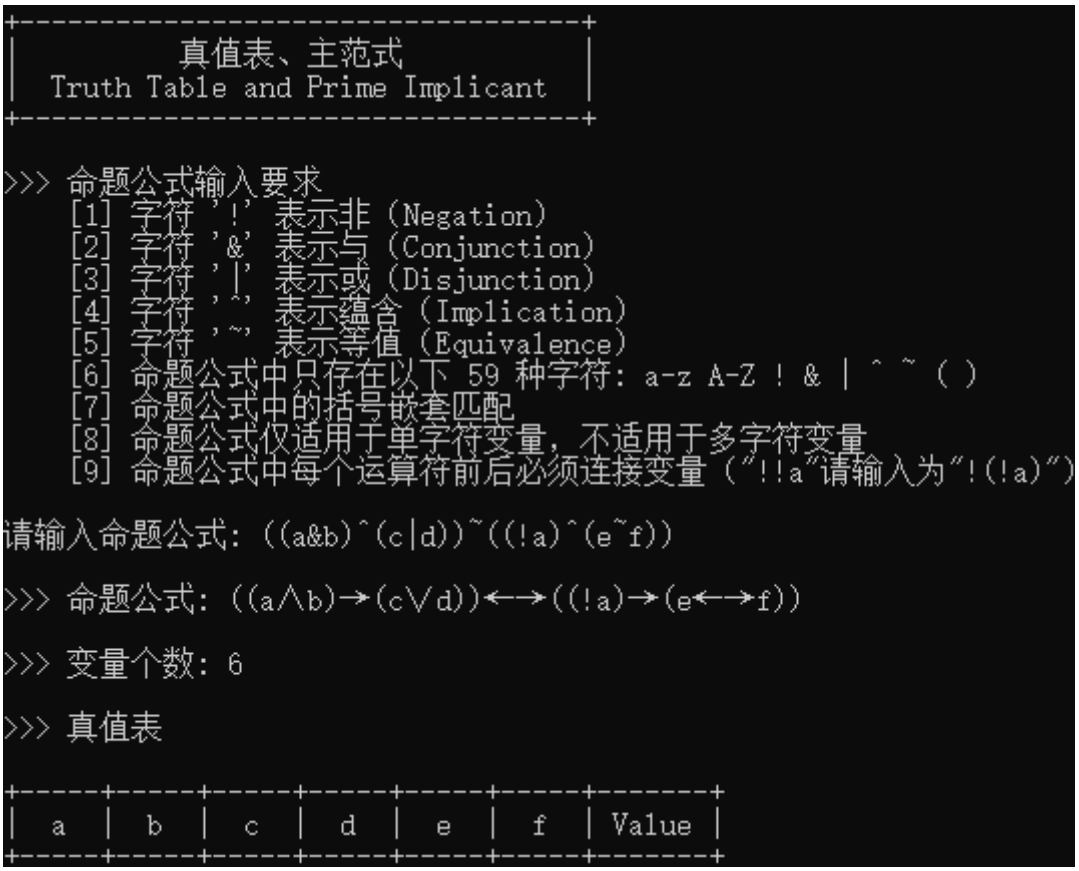
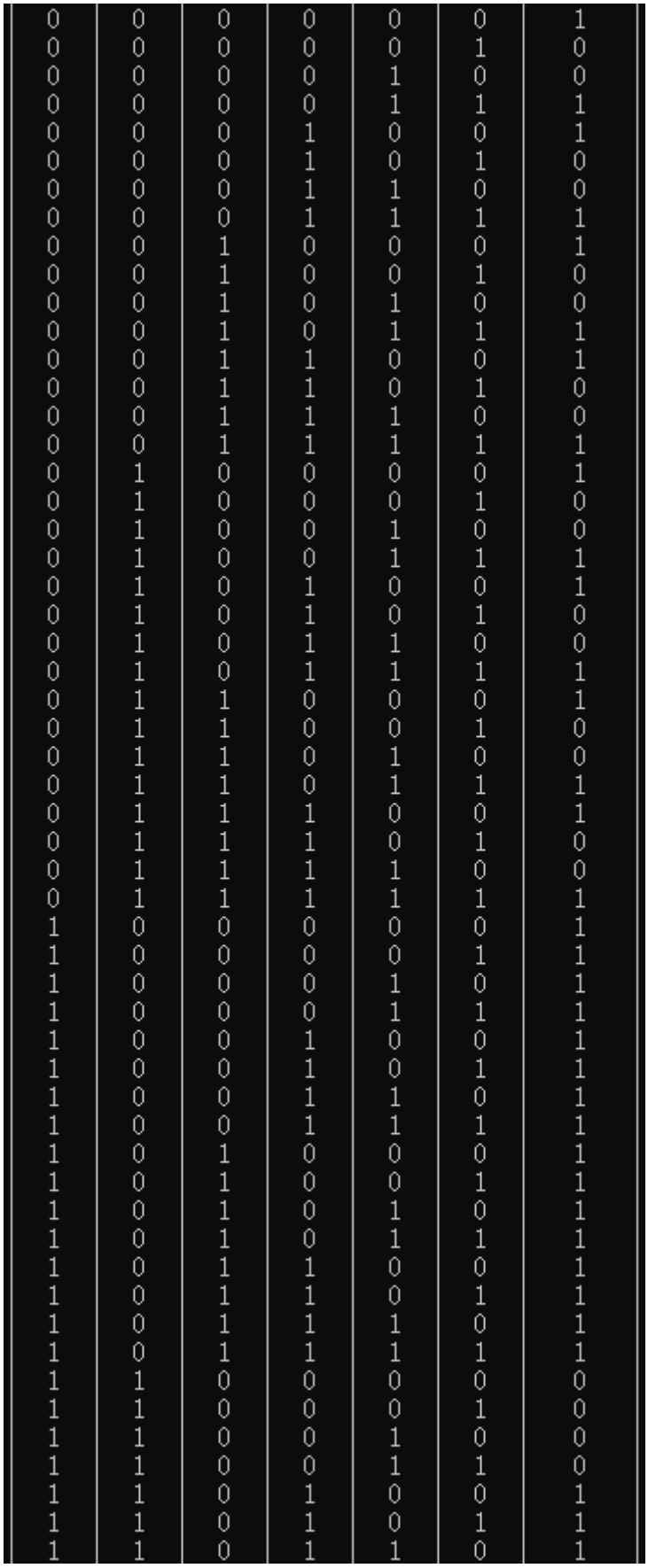


图 7.2.9 实验数据 9（第一部分）



0	0	0	0	0	0	1
0	0	0	0	0	1	0
0	0	0	0	1	0	0
0	0	0	0	1	1	1
0	0	0	1	0	0	1
0	0	0	1	0	1	0
0	0	0	1	1	0	0
0	0	0	1	1	1	1
0	0	1	0	0	0	1
0	0	1	0	0	1	0
0	0	1	0	1	0	0
0	0	1	0	1	1	1
0	0	1	1	0	0	1
0	0	1	1	0	1	0
0	0	1	1	1	0	0
0	0	1	1	1	1	1
0	1	0	0	0	0	1
0	1	0	0	0	1	0
0	1	0	0	1	0	0
0	1	0	0	1	1	1
0	1	0	1	0	0	1
0	1	0	1	0	1	0
0	1	0	1	1	0	0
0	1	0	1	1	1	1
0	1	1	0	0	0	1
0	1	1	0	0	1	0
0	1	1	0	1	0	0
0	1	1	0	1	1	1
0	1	1	1	0	0	1
0	1	1	1	0	1	0
0	1	1	1	1	0	0
0	1	1	1	1	1	1
1	0	0	0	0	0	1
1	0	0	0	0	1	1
1	0	0	0	1	0	1
1	0	0	0	1	1	1
1	0	0	1	0	0	1
1	0	0	1	0	1	1
1	0	0	1	1	0	1
1	0	0	1	1	1	1
1	0	1	0	0	0	1
1	0	1	0	0	1	1
1	0	1	0	1	0	1
1	0	1	0	1	1	1
1	0	1	1	0	0	1
1	0	1	1	0	1	1
1	0	1	1	1	0	1
1	0	1	1	1	1	1
1	1	0	0	0	0	0
1	1	0	0	0	1	0
1	1	0	0	1	0	0
1	1	0	0	1	1	0
1	1	0	1	0	0	1
1	1	0	1	0	1	1
1	1	0	1	1	0	1

图 7.2.10 实验数据 9（第二部分）

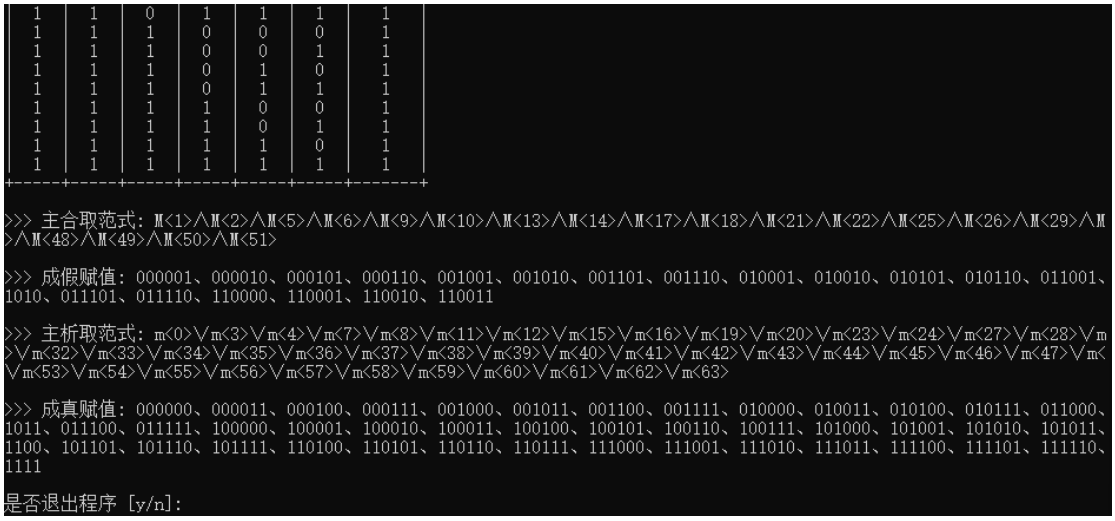


图 7.2.11 实验数据 9（第三部分）

实验数据分析 9：本实验用于测试非（Negation）、等值（Equivalence）、或（Disjunction）、蕴含（Implication）、等值（Equivalence）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))←→((!a)→(e←→f))”，变量个数为 6，真值表输出正确。

主合取范式：

M<1>∧M<2>∧M<5>∧M<6>∧M<9>∧M<10>∧M<13>∧M<14>∧M<17>∧M<18>∧M<21>∧M<22>∧M<25>∧M<26>∧M<29>∧M<30>∧M<48>∧M<49>∧M<50>∧M<51>。

成假赋值：

000001、000010、000101、000110、001001、001010、001101、001110、010001、010010、010101、010110、011001、011010、011101、011110、110000、110001、110010、110011。

主析取范式：

m<0>∨m<3>∨m<4>∨m<7>∨m<8>∨m<11>∨m<12>∨m<15>∨m<16>∨m<19>∨m<20>∨m<23>∨m<24>∨m<27>∨m<28>∨m<31>∨m<32>∨m<33>∨m<34>∨m<35>∨m<36>∨m<37>∨m<38>∨m<39>∨m<40>∨m<41>∨m<42>∨m<43>∨m<44>∨m<45>∨m<46>∨m<47>∨m<52>∨m<53>∨m<54>∨m<55>∨m<56>∨m<57>∨m<58>∨m<59>∨m<60>∨m<61>∨m<62>∨m<63>。

成真赋值：

000000、000011、000100、000111、001000、001011、001100、001111、010000、010011、010100、010111、011000、011011、011100、011111、100000、100001、100010、100011、100100、100101、100110、100111、101000、101001、101010、101011、101100、101101、101110、101111、110100、110101、110110、110111、111000、111001、111010、111011、111100、111101、111110、111111。

8 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（DNF）和主合取范式（CNF）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。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。

9 程序源文件

9.1 命题逻辑联接词

```
/*****************************************************************
 * Project Name:  Assignment_1_1
 * File Name:     assignment_1_1.cpp
 * File Function: 命题逻辑联接词
 * Author:        Jishen Lin (林继申)
 * Update Date:   2023/12/13
 *****************************************************************/

#include <iostream>
#include <conio.h>

/*
```

```
 * Function Name:    inputLogicalValue
 * Function:         Input logical value
 * Input Parameters: char falseValue
 *                   char trueValue
 * Return Value:     true / false
 */
bool inputLogicalValue(char falseValue = '0', char trueValue = '1')
{
    while (true) {
        char optn = _getch();
        if (optn == 0 || optn == -32)
            optn = _getch();
        else if (optn == falseValue || optn == trueValue) {
            std::cout << optn << std::endl << std::endl;
            return optn == falseValue ? false : true;
        }
    }
}

/*
 * Function Name:    main
 * Function:         Main function
 * Return Value:     0
 */
int main()
{
    do {
        /* System entry prompt */
        system("cls");
        std::cout << "+----------------------------------+" <<
std::endl;
        std::cout << "|          命题逻辑联接词          |" << std::endl;
        std::cout << "|  Propositional Logic Connectives  |" <<
std::endl;
        std::cout << "+----------------------------------+" <<
std::endl << std::endl;

        /* Enter the value of P and Q */
        std::cout << "请输入 P 的值 [0/1]: ";
        bool p = inputLogicalValue();
        std::cout << "请输入 Q 的值 [0/1]: ";
        bool q = inputLogicalValue();

        /* Propositional logic connectives */
```

```
        std::cout << ">>> 合取: P∧Q = " << (p && q) << std::endl <<
std::endl; // Conjunction
        std::cout << ">>> 析取: P∨Q = " << (p || q) << std::endl <<
std::endl; // Disjunction
        std::cout << ">>> 条件: P→Q = " << (!p || q) << std::endl <<
std::endl; // Implication
        std::cout << ">>> 双向条件: P↔Q = " << ((!p || q) && (!q || p))
<< std::endl << std::endl; // Equivalence

        /* Whether to exit the program */
        std::cout << "是否退出程序 [y/n]: ";
    } while (!inputLogicalValue('n', 'y'));
    return 0;
}
```

9.2 真值表、主范式

```
/*****************************************************************
 * Project Name:  Assignment_1_2
 * File Name:     assignment_1_2.cpp
 * File Function: 真值表、主范式
 * Author:        Jishen Lin (林继申)
 * Update Date:   2023/12/21
 *****************************************************************/

#include <iostream>
#include <string>
#include <vector>
#include <stack>
#include <queue>
#include <set>
#include <map>
#include <cmath>
#include <cctype>
#include <conio.h>
#include <algorithm>

/*
 * Function Name:    extractVariables
 * Function:         Split expression into unique variables
 * Input Parameters: const std::string& expression
 * Return Value:     variables
 */
std::set<char> extractVariables(const std::string& expression)
```

```
{
    std::set<char> variables;
    for (char ch : expression)
        if (isalpha(ch))
            variables.insert(ch);
    return variables;
}

/*
 * Function Name:    isOperator
 * Function:         Determine whether a character is an operator
 * Input Parameters: char ch
 * Return Value:     true / false
 */
bool isOperator(char ch)
{
    return ch == '!' || ch == '&' || ch == '|' || ch == '^' || ch ==
'~';
}

/*
 * Function Name:    getPrecedence
 * Function:         Get precedence
 * Input Parameters: char op
 * Return Value:     precedence
 */
int getPrecedence(char op)
{
    switch (op) {
        case '!': // Negation
            return 3;
        case '&': // Conjunction
        case '|': // Disjunction
            return 2;
        case '^': // Implication
        case '~': // Equivalence
            return 1;
        default: // Default
            return 0;
    }
}

/*
 * Function Name:    applyOperator
```

```
 * Function:         Apply operator
 * Input Parameters: char op
 *                   bool a
 *                   bool b
 * Return Value:     true / false
 */
bool applyOperator(char op, bool a, bool b = false)
{
    switch (op) {
        case '!':
            return !a; // Negation
        case '&':
            return a && b; // Conjunction
        case '|':
            return a || b; // Disjunction
        case '^':
            return !a || b; // Implication
        case '~':
            return a == b; // Equivalence
        default:
            exit(-1);
    }
}

/*
 * Function Name:    infixToPostfix
 * Function:         Convert an infix mathematical expression to its
corresponding postfix notation
 * Input Parameters: const std::string& expression
 * Return Value:     a queue containing the postfix notation of the
input expression
 */
std::queue<char> infixToPostfix(const std::string& expression)
{
    std::stack<char> ops; // Stack to hold operators
    std::queue<char> postfix; // Queue to hold postfix expression
    for (char ch : expression) {
        if (isalpha(ch)) {
            // If character is an operand (variable), add it to the
postfix queue
            postfix.push(ch);
        }
        else if (ch == '(') {
```

```
            // If character is an opening parenthesis, push it onto the
stack
            ops.push(ch);
        }
        else if (ch == ')') {
            // If character is a closing parenthesis, pop all operators
from the stack
            // and add to postfix queue until an opening parenthesis is
encountered
            while (!ops.empty() && ops.top() != '(') {
                postfix.push(ops.top());
                ops.pop();
            }
            ops.pop(); // Pop the opening parenthesis from the stack
        }
        else if (isOperator(ch)) {
            // If character is an operator, pop operators from the stack
            // with higher or equal precedence and add to postfix queue
            while (!ops.empty() && getPrecedence(ops.top()) >=
getPrecedence(ch)) {
                postfix.push(ops.top());
                ops.pop();
            }
            ops.push(ch); // Push the current operator onto the stack
        }
    }
    while (!ops.empty()) {
        // After reading the expression, pop any remaining
        // operators from the stack and add to postfix queue
        postfix.push(ops.top());
        ops.pop();
    }
    return postfix; // Return the postfix expression
}

/*
 * Function Name:    calculatePostfix
 * Function:         Calculate a given postfix notation expression
 * Input Parameters: std::queue<char>& postfix
 *                   const std::map<char, bool>& values
 * Return Value:     true / false
 */
bool calculatePostfix(std::queue<char>& postfix, const std::map<char,
bool>& values)
```

```
{
    std::stack<bool> s; // Stack to hold boolean values during
calculation
    while (!postfix.empty()) {
        char ch = postfix.front();
        postfix.pop();
        if (isalpha(ch))
            s.push(values.at(ch)); // If the character is an operand,
push its boolean value onto the stack
        else {
            if (ch == '!') {
                bool a = s.top(); // Pop the top value
                s.pop(); // Remove the top value from the stack
                s.push(!a); // Apply the NOT operation and push the
result back onto the stack
            }
            else {
                bool b = s.top(); // Pop the first (right operand) value
                s.pop(); // Remove the value from the stack
                bool a = s.top(); // Pop the second (left operand) value
                s.pop(); // Remove the value from the stack
                s.push(applyOperator(ch, a, b)); // Apply the binary
operation and push the result back onto the stack
            }
        }
    }
    return s.top(); // Return the final result from the top of the stack
}

/*
 * Function Name:    calculateExpression
 * Function:         Calculate expression
 * Input Parameters: const std::string& expression
 *                   const std::map<char, bool>& values
 * Return Value:     true / false
 */
bool calculateExpression(const std::string& expression, const
std::map<char, bool>& values)
{
    /* Convert an infix mathematical expression to its corresponding
postfix notation */
    auto postfix = infixToPostfix(expression);

    /* Calculate a given postfix notation expression */
```

```
    return calculatePostfix(postfix, values);
}

/*
 * Function Name:    printBinary
 * Function:         Print binary number with specified number of bits
 * Input Parameters: int number
 *                   int bits
 * Return Value:     void
 */
void printBinary(int number, int bits)
{
    for (int i = bits - 1; i >= 0; i--)
        std::cout << ((number >> i) & 1);
}

/*
 * Function Name:    outputTruthTableAndNormalForms
 * Function:         Output truth table and normal forms (CNF and DNF)
 * Input Parameters: const std::string& expression
 * Return Value:     void
 */
void outputTruthTableAndNormalForms(const std::string& expression)
{
    /* Function to print the separator line in the truth table */
    auto printSeparator = [](int numVariables) {
        for (int i = 0; i < numVariables; i++)
            std::cout << "+-----";
        std::cout << "+-------+" << std::endl;
        };

    /* Extract unique variables from the expression */
    auto variables = extractVariables(expression);
    int numVariables = static_cast<int>(variables.size());
    std::cout << std::endl << ">>> 变量个数: " << numVariables <<
std::endl;

    /* Start printing the truth table */
    std::cout << std::endl << ">>> 真值表" << std::endl << std::endl;
    printSeparator(numVariables);

    /* Print the header row of the truth table */
    std::cout << "| ";
    for (char var : variables)
```

```
        std::cout << " " << var << "  | ";
    std::cout << "Value |" << std::endl;
    printSeparator(numVariables);

    /* Store the results of the expression for each combination of
variables */
    std::vector<int> results;
    for (int i = 0; i < std::pow(2, numVariables); i++) {
        /* Set the values for each variable and print the row in the
truth table */
        std::cout << "|  ";
        std::map<char, bool> values;
        int index = 0;
        for (char var : variables) {
            values[var] = ((i >> (numVariables - 1 - index++)) & 1) ==
1;
            std::cout << values[var] << "  |  ";
        }

        /* Calculate the result of the expression with the current set
of values */
        bool result = calculateExpression(expression, values);
        results.push_back(result ? 1 : 0);
        std::cout << " " << result << "   |" << std::endl;
    }
    printSeparator(numVariables);

    /* Print the main conjunctive normal form (CNF) */
    bool first = true;
    std::cout << std::endl << ">>> 主合取范式: ";
    bool isAlwaysTrue = std::find(results.begin(), results.end(), 0) ==
results.end(); // Check if the expression is always true
    if (isAlwaysTrue)
        std::cout << "1";
    else {
        /* For each false result, add a clause to the CNF */
        for (int i = 0; i < std::pow(2, numVariables); i++)
            if (results[i] == 0) {
                if (first) {
                    std::cout << "M<" << i << ">";
                    first = false;
                }
                else
                    std::cout << "∧M<" << i << ">";
```

```
            }
        }

        /* Print false sssignments */
        first = true;
        std::cout << std::endl << std::endl << ">>> 成假赋值: ";
        if (isAlwaysTrue)
            std::cout << "无";
        else {
            for (int i = 0; i < std::pow(2, numVariables); i++)
                if (results[i] == 0) {
                    if (first) {
                        printBinary(i, numVariables);
                        first = false;
                    }
                    else {
                        std::cout << "、";
                        printBinary(i, numVariables);
                    }
                }
        }

        /* Print the main disjunctive normal form (DNF) */
        first = true;
        std::cout << std::endl << std::endl << ">>> 主析取范式: ";
        bool isAlwaysFalse = std::find(results.begin(), results.end(), 1) ==
    results.end(); // Check if the expression is always false
        if (isAlwaysFalse)
            std::cout << "0";
        else {
            /* For each true result, add a clause to the DNF */
            for (int i = 0; i < std::pow(2, numVariables); i++)
                if (results[i] == 1) {
                    if (first) {
                        std::cout << "m<" << i << ">";
                        first = false;
                    }
                    else
                        std::cout << "∨m<" << i << ">";
                }
        }

        /* Print true sssignments */
        first = true;
```

```
    std::cout << std::endl << std::endl << ">>> 成真赋值: ";
    if (isAlwaysFalse)
        std::cout << "无";
    else {
        for (int i = 0; i < std::pow(2, numVariables); i++)
            if (results[i] == 1) {
                if (first) {
                    printBinary(i, numVariables);
                    first = false;
                }
                else {
                    std::cout << "、 ";
                    printBinary(i, numVariables);
                }
            }
    }
    std::cout << std::endl << std::endl;
}

/*
 * Function Name:    replaceAll
 * Function:         Replace all occurrences of a substring in a string
with another substring
 * Input Parameters: std::string str
 *                   const std::string& from
 *                   const std::string& to
 * Return Value:     string
 */
std::string replaceAll(std::string str, const std::string& from, const
std::string& to)
{
    size_t start_pos = 0;
    while ((start_pos = str.find(from, start_pos)) != std::string::npos)
{
        str.replace(start_pos, from.length(), to); // Replace from with
to at start_pos
        start_pos += to.length(); // Move past the end of the newly
inserted substring
    }
    return str;
}

/*
 * Function Name:    replaceSymbols
```

```
 * Function:         Replace specific symbol substrings in the input
string with their designated replacements
 * Input Parameters: std::string input
 * Return Value:     string
 */
std::string replaceSymbols(std::string input)
{
    input = replaceAll(input, "&", "∧"); // Replace "&" with "∧"
    input = replaceAll(input, "|", "∨"); // Replace "|" with "∨"
    input = replaceAll(input, "~", "↔"); // Replace "~" with "↔"
    input = replaceAll(input, "^", "→"); // Replace "^" with "→"
    return input;
}

/*
 * Function Name:    isValidExpression
 * Function:         Check if a expression is valid
 * Input Parameters: const std::string& expression
 * Return Value:     true / false
 */
bool isValidExpression(const std::string& expression)
{
    if (expression.empty()) {
        std::cout << ">>> 命题公式为空，请重新输入！" << std::endl;
        return false;
    }
    std::stack<char> parentheses;
    char previous = '\0';
    for (char ch : expression) {
        if (!((ch >= 'A' && ch <= 'Z') || (ch >= 'a' && ch <= 'z'))
&& !isOperator(ch) && ch != '(' && ch != ')') {
            std::cout << ">>> 命题公式存在非法字符输入，请重新输入！" <<
std::endl;
            return false;
        }
        if (previous == '\0' && (ch == '&' || ch == '|' || ch == '~' ||
ch == '^')) {
            std::cout << ">>> 命题公式不能以二元运算符开始，请重新输入！" <<
std::endl;
            return false;
        }
        if (previous == '(' && ch == ')') {
            std::cout << ">>> 命题公式存在空括号，请重新输入！" <<
std::endl;
```

```
            return false;
        }
        if (previous == '!' && ch == '!') {
            std::cout << ">>> 命题公式存在不合法的连续取非操作，请重新输入！
" << std::endl;
            return false;
        }
        if (ch == '!' && isalpha(previous)) {
            std::cout << ">>> 命题公式中取非运算符前不可连接变量，请重新输
入！" << std::endl;
            return false;
        }
        if ((isalpha(ch) && previous == ')') || (ch == '(' &&
isalpha(previous))) {
            std::cout << ">>> 命题公式中变量与括号的连接不正确，请重新输入！
" << std::endl;
            return false;
        }
        if (ch == '(')
            parentheses.push(ch);
        else if (ch == ')') {
            if (parentheses.empty()) {
                std::cout << ">>> 命题公式括号不匹配，请重新输入！" <<
std::endl;
                return false;
            }
            parentheses.pop();
        }
        if (isalpha(ch) && isalpha(previous)) {
            std::cout << ">>> 命题公式仅适用于单字符变量，不适用于多字符变
量，请重新输入！" << std::endl;
            return false;
        }
        if ((ch == '&' || ch == '|' || ch == '~' || ch == '^') &&
(!isalpha(previous) && previous != ')')) {
            std::cout << ">>> 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;
            return false;
        }
        previous = ch;
    }
    if (previous == '&' || previous == '|' || previous == '~' ||
previous == '^' || previous == '!') {
```

```
        std::cout << ">>> 命题公式不能以运算符结尾，请重新输入！" <<
std::endl;
        return false;
    }
    if (!parentheses.empty()) {
        std::cout << ">>> 命题公式括号不匹配，请重新输入！" << std::endl;
        return false;
    }
    return true;
}

/*
 * Function Name:    inputLogicalValue
 * Function:         Input logical value
 * Input Parameters: char falseValue
 *                   char trueValue
 * Return Value:     true / false
 */
bool inputLogicalValue(char falseValue = '0', char trueValue = '1')
{
    while (true) {
        char optn = _getch();
        if (optn == 0 || optn == -32)
            optn = _getch();
        else if (optn == falseValue || optn == trueValue) {
            std::cout << optn << std::endl << std::endl;
            return optn == falseValue ? false : true;
        }
    }
}

/*
 * Function Name:    main
 * Function:         Main function
 * Return Value:     0
 */
int main()
{
    do {
        /* System entry prompt */
        system("cls");
        std::cout << "+----------------------------------+" <<
std::endl;
        std::cout << "|          真值表、主范式          |" << std::endl;
```

```
        std::cout << "|  Truth Table and Prime Implicant  |" <<
std::endl;
        std::cout << "+-----------------------------------+" <<
std::endl << std::endl;
        std::cout << ">>> 命题公式输入要求" << std::endl;
        std::cout << "    [1] 字符 '!' 表示非（Negation）" << std::endl;
        std::cout << "    [2] 字符 '&' 表示与（Conjunction）" <<
std::endl;
        std::cout << "    [3] 字符 '|' 表示或（Disjunction）" <<
std::endl;
        std::cout << "    [4] 字符 '^' 表示蕴含（Implication）" <<
std::endl;
        std::cout << "    [5] 字符 '~' 表示等值（Equivalence）" <<
std::endl;
        std::cout << "    [6] 命题公式中只存在以下 59 种字符: a-z A-Z ! & |
^ ~ ( )" << std::endl;
        std::cout << "    [7] 命题公式中的括号嵌套匹配" << std::endl;
        std::cout << "    [8] 命题公式仅适用于单字符变量，不适用于多字符变量
" << std::endl;
        std::cout << "    [9] 命题公式中每个运算符前后必须连接变量（\"!!a\"
请输入为\"!(!a)\"）" << std::endl;

        /* Input a a propositional formula */
        std::string expression;
        do {
            std::cout << std::endl << "请输入命题公式: ";
            std::cin >> expression;
            std::cout << std::endl;
        } while (!isValidExpression(expression));
        std::cout << ">>> 命题公式: " << replaceSymbols(expression) <<
std::endl;

        /* Output truth table and normal forms (CNF and DNF) */
        outputTruthTableAndNormalForms(expression);

        /* Whether to exit the program */
        std::cout << "是否退出程序 [y/n]: ";
    } while (!inputLogicalValue('n', 'y'));
    return 0;
}
```