Практическая часть

Разработать программу, реализующую метод резолюций для логики высказываний. Программа считывает несколько формул-посылок и следствие, истинность которого необходимо доказать. На экран программа выводит не только результат доказательства (фразы «Теорема доказана», «Теорема опровергнута»), но и объяснение доказательства, а именно исходное множество дизъюнктов и выполненные склейки.

```
#include <iostream>
#include <string>
#include <set>
#include <map>
#include <stack>
#include <queue>
#include <cassert>
#include <ctime>
#include <algorithm>
#include <windows.h>

// Объявление типов.
// Токен (лексема):
typedef char Token;
// Стек токенов:
typedef std::stack<Token> Stack;
// Последовательность токенов:
typedef std::queue<Token> Queue;
// Множество различных токенов:
typedef std::set<Token> Set;
// Таблица значений переменных:
typedef std::map<Token, Token> Map;
// Пара переменная—значение:
typedef std::pair<Token, Token> VarVal;
// Строка символов:
typedef std::string String;

// Является ли токен числом?
inline bool isNumber(Token t) {
  return t == '0' || t == '1';
}

// Является ли токен переменной?
inline bool isVariable(Token t) {
  return (t >= 'A' && t <= 'Z') || (t >= 'a' && t <= 'z');
}

// Является ли токен операцией?
inline bool isOperation(Token t) {
  return (t == '|' || t == '&' || t == '-' || t == '>' || t == '~');
}

// Является ли токен открывающей скобкой?
inline bool isOpeningPar(Token t) {
  return t == '(';
}

// Является ли токен закрывающей скобкой?
inline bool isClosingPar(Token t) {
  return t == ')';
}

// Вернуть величину приоритета операции
// (чем больше число, тем выше приоритет)
```

```
inline int priority(Token op) {
  assert  (isOperation(op));
  int res = 0;
  switch (op) {
    case '-':
      // Отрицание — наивысший приоритет
      res = 5;
      break;
    case '&':
      // Конъюнкция
      res = 4;
      break;
    case '|':
      // Дизъюнкция
      res = 3;
      break;
    case '>':
      // Импликация
      res = 2;
      break;
    case '~':
      // Эквивалентность — наинизший приоритет
      res = 1;
      break;
  }
  return res;
}

// Преобразовать последовательность токенов,
// представляющих выражение в инфиксной записи,
// в последовательность токенов, представляющих
// выражение в обратной польской записи
// (алгоритм Дейкстры «Сортировочная станция»)
Queue infixToPostfix(Queue input) {
  // Выходная последовательность (очередь вывода):
  Queue output;
  // Рабочий стек:
  Stack s;
  // Текущий входной токен:
  Token t;
  // Пока есть токены во входной последовательности:
  while (!input.empty()) {
    // Получить токен из начала входной последовательности
    t = input.front();
    input.pop();
    // Если токен — число или переменная, то:
    if (isNumber(t) || isVariable(t)) {
      // Добавить его в очередь вывода
      output.push(t);
      // Если токен — операция op1, то:
    } else if (isOperation(t)) {
      // Пока на вершине стека присутствует токен-операция op2
      // и у op1 приоритет меньше либо равен приоритету op2, то:
      while (!s.empty() && isOperation(s.top())
             && priority(t) <= priority(s.top())
             ) {
        // переложить op2 из стека в выходную очередь
        output.push(s.top());
        s.pop();
      }
      // Положить op1 в стек
      s.push(t);
```

```
      // Если токен — открывающая скобка, то:
    } else if (isOpeningPar(t)) {
      // Положить его в стек
      s.push(t);
      // Если токен — закрывающая скобка, то:
    } else if (isClosingPar(t)) {
      // Пока токен на вершине стека не является открывающей скобкой:
      while (!s.empty() && !isOpeningPar(s.top())) {
        // Перекладывать токены-операции из стека
        // в выходную очередь
        assert  (isOperation(s.top()));
        output.push(s.top());
        s.pop();
      }
      // Если стек закончился до того,
      // как был встречен токен-«открывающая скобка», то:
      if (s.empty()) {
        // В выражении пропущена открывающая скобка
        throw String("Пропущена открывающая скобка!");
      } else {
        // Иначе выкинуть открывающую скобку из стека
        // (но не добавлять в очередь вывода)
        s.pop();
      }
    } else {
      // В остальных случаях входная последовательность
      // содержит токен неизвестного типа
      String msg("Неизвестный символ \'");
      msg += t + String("\'!");
      throw msg;
    }
  }
  // Токенов на входе больше нет, но ещё могут остаться токены в стеке.
  // Пока стек не пустой:
  while (!s.empty()) {
    // Если токен на вершине стека — открывающая скобка, то:
    if (isOpeningPar(s.top())) {
      // В выражении присутствует незакрытая скобка
      throw String("Незакрытая скобка!");
    } else {
      // Иначе переложить токен-операцию из стека в выходную очередь
      assert  (isOperation(s.top()));
      output.push(s.top());
      s.pop();
    }
  }
  // Конец алгоритма.
  // Выдать полученную последовательность
  return output;
}

// Напечатать последовательность токенов
void printSequence(Queue q) {
  while (!q.empty()) {
    std::cout << q.front();
    q.pop();
  }
  std::cout << std::endl;
}

// Является ли символ пробельным?
inline bool isSpace(char c) {
```

```
  return c <= ' ';
}

// Если символ — маленькая буква, преобразовать её в большую,
// иначе просто вернуть этот же символ
inline char toUpperCase(char c) {
  if (c >= 'a' && c <= 'z') {
    return c - 'a' + 'A';
  } else {
    return c;
  }
}

// Преобразовать строку с выражением в последовательность токенов
// (лексический анализатор)
Queue stringToSequence(const String &s) {
  Queue res;
  for (char i : s) {
    if (!isSpace(i)) {
      res.push(toUpperCase(i));
    }
  }
  return res;
}

// Напечатать сообщение об ошибке
inline void printErrorMessage(const String &err) {
  std::cerr << "*** ОШИБКА! " << err << std::endl;
}

// Ввести выражение с клавиатуры
inline String inputExpr() {
  String expr;
  std::cout << "Формула логики высказываний: ";
  std::getline(std::cin, expr);
  return expr;
}

// Выделить из последовательности токенов переменные
Set getVariables(Queue s) {
  Set res;
  while (!s.empty()) {
    if (isVariable(s.front()) && res.count(s.front()) == 0) {
      res.insert(s.front());
    }
    s.pop();
  }
  return res;
}

// Получить значения переменных с клавиатуры
Map inputVarValues(const Set &var) {
  Token val;
  Map res;
  for (Set::const_iterator i = var.begin(); i != var.end(); ++i) {
    do {
      std::cout << *i << " = ";
      std::cin >> val;
      if (!isNumber(val)) {
        std::cerr << "Введите 0 или 1!" << std::endl;
      }
    } while (!isNumber(val));
    res.insert(VarVal(*i, val));
```

```
  }
  return res;
}

// Заменить переменные их значениями
Queue substValues(Queue expr, Map &varVal) {
  Queue res;
  while (!expr.empty()) {
    if (isVariable(expr.front())) {
      res.push(varVal[expr.front()]);
    } else {
      res.push(expr.front());
    }
    expr.pop();
  }
  return res;
}

// Является ли операция бинарной?
inline bool isBinOp(Token t) {
  return t == '&' || t == '|' || t == '>' || t == '~';
}

// Является ли операция унарной?
inline bool isUnarOp(Token t) {
  return t == '-';
}

// Получить bool-значение токена-числа (true или false)
inline bool logicVal(Token x) {
  assert (isNumber(x));
  return x == '1';
}

// Преобразовать bool-значение в токен-число
inline Token boolToToken(bool x) {
  if (x) {
    return '1';
  } else {
    return '0';
  }
}

// Вычислить результат бинарной операции
inline Token evalBinOp(Token a, Token op, Token b) {
  assert (isNumber(a) && isBinOp(op) && isNumber(b));
  bool res;
  // Получить bool-значения операндов
  bool left = logicVal(a);
  bool right = logicVal(b);
  switch (op) {
    case '&':
      // Конъюнкция
      res = left && right;
      break;
    case '|':
      // Дизъюнкция
      res = left || right;
      break;
    case '>':
      // Импликация
      res = !left || right;
```

```
      break;
    case '~':
      // Эквивалентность
      res = (!left || right) && (!right || left);
      break;
  }
  return boolToToken(res);
}

// Вычислить результат унарной операции
inline Token evalUnarOp(Token op, Token a) {
  assert  (isUnarOp(op) && isNumber(a));
  bool res = logicVal(a);
  switch (op) {
    case '-':
      // Отрицание
      res = !res;
      break;
  }
  return boolToToken(res);
}

// Вычислить значение операции, модифицируя стек.
// Результат помещается в стек
void evalOpUsingStack(Token op, Stack &s) {
  assert  (isOperation(op));
  // Если операция бинарная, то:
  if (isBinOp(op)) {
    // В стеке должны быть два операнда
    if (s.size() >= 2) {
      // Если это так, то извлекаем правый операнд-число
      Token b = s.top();
      if (!isNumber(b)) {
        throw String("Неверное выражение!");
      }
      s.pop();
      // Затем извлекаем левый операнд-число
      Token a = s.top();
      if (!isNumber(a)) {
        throw String("Неверное выражение!");
      }
      s.pop();
      // Помещаем в стек результат операции
      s.push(evalBinOp(a, op, b));
    } else {
      throw String("Неверное выражение!");
    }
    // Иначе операция унарная
  } else if (isUnarOp(op) && !s.empty()) {
    // Извлекаем операнд
    Token a = s.top();
    if (!isNumber(a)) {
      throw String("Неверное выражение!");
    }
    s.pop();
    // Помещаем в стек результат операции
    s.push(evalUnarOp(op, a));
  } else {
    throw String("Неверное выражение!");
  }
}
```

```
// Вычислить значение выражения, записанного в обратной польской записи
Token evaluate(Queue expr) {
  // Рабочий стек
  Stack s;
  // Текущий токен
  Token t;
  // Пока входная последовательность содержит токены:
  while (!expr.empty()) {
    // Считать очередной токен
    t = expr.front();
    assert  (isNumber(t) || isOperation(t));
    expr.pop();
    // Если это число, то:
    if (isNumber(t)) {
      // Поместить его в стек
      s.push(t);
      // Если это операция, то:
    } else if (isOperation(t)) {
      // Вычислить её, модифицируя стек
      // (результат также помещается в стек)
      evalOpUsingStack(t, s);
    }
  }
  // Результат — единственный элемент в стеке
  if (s.size() == 1) {
    // Вернуть результат
    return s.top();
  } else {
    throw String("Неверное выражение!");
  }
}

// Вывести результат вычисления на экран
void printResult(Token r) {
  assert  (isNumber(r));
  std::cout << "Значение выражения: " << r << std::endl;
}

/*Возвращает словарь, элементы которого имеют вид [переменная: значение] или
[var[i] : a[i]].*/
Map input_by_arr(const Set &vars, int *a) {
  Token val;
  Map res;
  for (auto i = vars.begin(); i != vars.end(); i++) { // перебираем множество переменных
    val = (*a ? '1' : '0'); // подготавливаем значение в соответствии с массивом a
    a++;
    if (!isNumber(val)) {
      std::cerr << "Введите 0 или 1!" << std::endl;
    }
    res.insert(VarVal(*i, val)); // вносим в список пару «переменная : значение»
  }
  return res;
}

/*Строим таблицу истинности truth_table для формулы input, имеющую переменные
vars.*/
void BuildTruthTable(Queue &input, Set &vars, std::vector<std::vector<int>>
&truth_table) {
  static int binary_arr[100];
  static int i = 0; // счетчик бинарного вектора
  static int z = 0; // счетчик строки матрицы truth_table (таблицы истинности)
  for (int x = 0; x < 2; x++) { // порождение всех бинарных векторов длины = количеству переменных
```

```
    binary_arr[i] = x;
    if (i == vars.size() - 1) { //бинарный вектор построен
      Map map_vars = input_by_arr(vars, binary_arr); // создаем список пар «переменная»
      Token r = evaluate(substValues(input, map_vars)); // вычисляем значение формулы при заданных переменных
      // заносим данные в таблицу истинности truth_table
      for (int k = 0; k <= i; k++)
        truth_table[z][k] = binary_arr[k];
      truth_table[z][vars.size()] = (r == '1');
      z++;
    } else {
      i++;
      BuildTruthTable(input, vars, truth_table);
      i--;
    }
  }
  if (i == 0) {
    z = 0;
  }
}

/*Структура - дизъюнкт.
i-й элемент вектора sign соответствует i+1 переменной множества всех
переменных.
Состояния i-ого элемента вектора:
1 – в дизъюнкте есть i-ая переменной
0 – в дизъюнкте нет i-ой переменной
-1 –в дизъюнкте есть отрицание i-ой переменной */
typedef struct Disjunct {
  std::vector<int> sign;
} Disjunct;

/*Возвращает структуру-дизъюнкт, построенную по вектору a.*/
Disjunct CreateDisjunct(std::vector<int> &a) {
  Disjunct res = {a};
  return res;
}

// /*Возвращает значениен "истина", если в векторе a есть элемент x,
// иначе возвращает значение "ложь".*/
int FindInVector(std::vector<int> &a, int x) {
  return find(a.begin(), a.end(), x) != a.end();
}

/* Возвращает значение "истина", если в векторе дизъюнктов a есть дизъюнкт x,
иначе возвращает значение "ложь".*/
int FindInVector(std::vector<Disjunct> &a, std::vector<int> &x) {
  for (int i = 0; i < a.size(); i++) {
    int j = 0;
    for (j = 0; j < x.size(); j++)
      if (a[i].sign[j] != x[j])
        break;
    if (j == x.size())
      return 1;
  }
  return 0;
}

/*Вывод дизъюнкта a на экран. Используются переменные из множества vars.*/
void PrintDisjunct(std::vector<int> &a, Set &vars) {
  bool fl_first = 0; //флаг того, что первый элемент выведен
  int i = 0;
  auto iter = vars.begin();
  //выводим до первого элемента включительно
```

```
  std::cout << "(";
  for (; !fl_first and i < a.size() and iter != vars.end(); i++, iter++)
    if (a[i] != 0) {
      fl_first = 1;
      if (a[i] == -1)
        std::cout << '-';
      std::cout << *iter << " ";
    }
  // продолжение вывода
  for (; i < a.size() and iter != vars.end(); i++, iter++)
    if (a[i] != 0) {
      std::cout << "| "; //отличие – добавление знака дизъюнкции
      if (a[i] == -1)
        std::cout << '-';
      std::cout << *iter << " ";
    }
  std::cout << ")";
}

/*Заполняет массив дизъюнктов a дизъюнктами СКНФ, относящейся к таблице
истинности truth_table.*/
void GetDisjunctArr(std::vector<Disjunct> &a, std::vector<std::vector<int>>
&truth_table) {
  for (int i = 0; i < truth_table.size(); i++)
    if (truth_table[i][truth_table[i].size() - 1] == 0) {
      a.resize(a.size() + 1);
      a[a.size() - 1].sign.resize(truth_table[i].size() - 1);
      for (int j = 0; j < truth_table[i].size() - 1; j++)
        a[a.size() - 1].sign[j] = truth_table[i][j] == 1 ? -1 : 1;
    }
}

/*Вывод массива дизъюнктов a. Используются переменные из множества vars. */
void PrintDisjunctArr(std::vector<Disjunct> &a, Set &vars) {
  std::cout << "Множество дизъюнктов: {";
  for (int i = 0; i < a.size(); i++) {
    PrintDisjunct(a[i].sign, vars);
    if (i != a.size() - 1)
      std::cout << ", ";
  }
  std::cout << "}\n";
}

/*Создает резольвенту res на основе дизъюнкции dis1 и dis 2. k-ая переменная
сокращается.*/
void GetResolvent(std::vector<int> &res, std::vector<int> &dis1,
                  std::vector<int> &dis2, int k) {
  res = dis1;
  for (int i = 0; i < dis2.size(); i++)
    if (res[i] == 0 and dis2[i] != 0)
      res[i] = dis2[i];
  res[k] = 0;
}

/*dis_arr – массив дизъюнктантов; vars – множество используемых переменных.
Поиск новых резольвент. Возвращает 2, если из массива dis_arr можно получить пустую резольвенту,
используя принцип резолюции. Возвращает 1, если из массива dis_arr можно нельзя пустую резольвенту,
используя принцип резолюции. Возвращает 0, если при использовании принципа резолюции для получения
резольвент произошло зацикливание.*/
int FindResolvents(std::vector<Disjunct> &dis_arr, Set &vars) {
  const time_t MAX_TIME = 19000; // максимальное время, которое отводится на работу цикла
  time_t start = time(NULL); // время начала поиска решения
```

```
  int fl_condition = 0; // исходное значение флага-состояния
  while (time(NULL) - start < MAX_TIME and fl_condition == 0) {
    bool there_is_new_resolvent = false; // на данной итерации была получена новая резольвента?
    // сформируем все возможные резольвенты на данной итерации
    for (int i = 0; i < dis_arr.size(); i++)
      for (int j = i + 1; j < dis_arr.size() and fl_condition == 0;
           j++) {
        std::vector<int> new_resolvent;
        // пытаемся создать резольвенту из i-го и j-го дизъюнктов
        for (int k = 0; k < vars.size(); k++)
          if (dis_arr[i].sign[k] != dis_arr[j].sign[k] and
              dis_arr[i].sign[k] != 0 &&
              dis_arr[j].sign[k] != 0) {
            GetResolvent(new_resolvent, dis_arr[i].sign,
                         dis_arr[j].sign, k);
            // если такой резольвенты нет в массиве дизъюнкт dis_arr – добавим и выведем вычисления
            if (FindInVector(dis_arr, new_resolvent) == 0) {
              there_is_new_resolvent = true; // получена новая резольвента
              PrintDisjunct(dis_arr[i].sign, vars);
              std::cout << " | ";
              PrintDisjunct(dis_arr[j].sign, vars);
              std::cout << " = ";
              PrintDisjunct(new_resolvent, vars);
              std::cout << '\n';
              dis_arr.push_back(CreateDisjunct(new_resolvent));
            }
            // если мы получили пустую резольвенту – выход. формула общезначима.
            if (FindInVector(new_resolvent, 1) == 0 and
                FindInVector(new_resolvent, -1) == 0) {
              fl_condition = 2;
              break;
            }
          }
      }
    // если за итерацию основного цикла не получено новых резольвент – выход. формула не общезначима.
    if (there_is_new_resolvent == 0) {
      std::cout << "Невозможно создать новую резольвенту.\n";
      fl_condition = 1;
    }
  }
  return fl_condition;
}

/* Метод резолюций для проверки формулы, записанной в посылках sends и
следствии consequence на общезначимость.
Возвращает 2, если формула общезначима.
Возвращает 1, если формула не общезначима.
Возвращает 0, если нельзя ничего сказать об общезначимости формулы.*/
int Resolution(String sends, String consequence) {
  // приводим формулу к виду противоречивости
  String formula = sends + "&" + "-" + "(" + consequence + ")";
  // Преобразуем строку в последовательность токенов
  Queue input = stringToSequence(formula);
  // Преобразовать последовательность токенов в ОПЗ
  Queue output = infixToPostfix(input);
  // Выделяем из токенов токены-переменные
  Set vars = getVariables(output);
  // Создаем матрицу truth_table (таблицу истинности) размера m на n, где m количество всех двоичных
векторов длины =
  // количеству переменных, n = количеств переменных + 1
  std::vector<std::vector<int>> truth_table(1 << vars.size());
  for (int i = 0; i < (1 << vars.size()); i++)
```

```
    truth_table[i].resize(vars.size() + 1);
  //заполняем таблицу истинности
  BuildTruthTable(output, vars, truth_table);
  // Получим наш массив дизъюнктов dis_arr по таблице истинности
  std::vector<Disjunct> dis_arr;
  GetDisjunctArr(dis_arr, truth_table);
  //выводим массив дизъюнктов
  PrintDisjunctArr(dis_arr, vars);
  //поиск новых резольвент
  return FindResolvents(dis_arr, vars);
}

int main() {
  SetConsoleOutputCP(CP_UTF8);
  std::cout << "Количество посылок: ";
  int n;
  std::cin >> n;
  std::cout << "\nПосылки:\n";
  String send1;
  std::cin >> send1;
  String sends = "(" + send1 + ")";
  for (int i = 1; i < n; i++) {
    std::cin >> send1;
    sends = sends + "&" + "(" + send1 + ")";
  }
  std::cout << "Следствие:\n";
  String consequence;
  std::cin >> consequence;
  switch (Resolution(sends, consequence)) {
    case 2:
      std::cout << "\nБыла получена пустая резольвента, выходит, следствие верное.\n";
      break;
    case 1:
      std::cout << "\nне была получена пустая резольвента, выходит, следствие не верное.\n";
      break;
    case 0:
      std::cout << "\nНевозможно получить ответ с помощью метода резолюций.\n";
      break;
  }
  return 0;
}
```

```
C:\BGTU\BGTU\MatLogika\3lab\Code\cmake-build-debug\Code.exe
Количество посылок:1

Посылки:
A|(A&B)
Следствие:
A
Множество дизъюнктов: {(A | B ), (A | -B ), (-A | B ), (-A | -B )}
(A | B ) | (A | -B ) = (A )
(A | B ) | (-A | B ) = (B )
(A | -B ) | (-A | B ) = (-B )
(-A | B ) | (-A | -B ) = (-A )
(A ) | (-A ) = ()

Была получена пустая резольвента, выходит, следствие верное.

Process finished with exit code 0
```

```
C:\BGTU\BGTU\MatLogika\3lab\Code\cmake-build-debug\Code.exe
Количество посылок:1

Посылки:
A&B
Следствие:
B&A
Множество дизъюнктов: {(A | B ), (A | -B ), (-A | B ), (-A | -B )}
(A | B ) | (A | -B ) = (A )
(A | B ) | (-A | B ) = (B )
(A | -B ) | (-A | B ) = (-B )
(-A | B ) | (-A | -B ) = (-A )
(A ) | (-A ) = ()

Была получена пустая резольвента, выходит, следствие верное.

Process finished with exit code 0
```

Вывод: в ходе лабораторной работы были изучены формальные теории, разработана программа, реализующая метод резолюций для логики высказываний