Министерство образования и науки РФ

Федеральное государственное автономное
образовательное учреждение высшего образования
«Национальный исследовательский университет ИТМО»

Факультет программной инженерии и компьютерной техники

Лабораторная работа 2

по дисциплине

«Системное программное обеспечение»

Выполнил:

Кустарев Иван Палович 309681

Преподаватель:

Кореньков Юрий Дмитриевич

Санкт-Петербург, 2025

Цель:

Реализовать построение графа потока управления посредством анализа дерева разбора для набора входных файлов. Выполнить анализ собранной информации и сформировать набор файлов с графическим представлением для результатов анализа.

Задачи:

Порядок выполнения:

1. Описать структуры данных, необходимые для представления информации о наборе файлов, наборе подпрограмм и графе потока управления, где:
    a. Для каждой подпрограммы: имя и информация о сигнатуре, граф потока управления, имя исходного файла с текстом подпрограммы.
    b. Для каждого узла в графе потока управления, представляющего собой базовый блок алгоритма подпрограммы: целевые узлы для безусловного и условного перехода (по мере необходимости), дерево операций, ассоциированных с данным местом в алгоритме, представленном в исходном тексте подпрограммы
2. Реализовать модуль, формирующий граф потока управления на основе синтаксической структуры текста подпрограмм для входных файлов
    a. Программный интерфейс модуля принимает на вход коллекцию, описывающую набор анализируемых файлов, для каждого файла – имя и соответствующее дерево разбора в виде структуры данных, являющейся результатом работы модуля, созданного по заданию 1 (п. 3.b).
    b. Результатом работы модуля является структура данных, разработанная в п. 1, содержащая информацию о проанализированных подпрограммах и коллекция с информацией об ошибках
    c. Посредством обхода дерева разбора подпрограммы, сформировать для неё граф потока управления, порождая его узлы и формируя между ними дуги в зависимости от синтаксической конструкции, представленной данным узлом дерева разбора: выражение, ветвление, цикл, прерывание цикла, выход из подпрограммы – для всех синтаксических конструкций по варианту (п. 2.b)
    d. С каждым узлом графа потока управления связать дерево операций, в котором каждая операция в составе текста программы представлена как совокупность вида операции и соответствующих операндов (см задание 1, пп. 2.d-g)
    e. При возникновении логической ошибки в синтаксической структуре при обходе дерева разбора, сохранить в коллекции информацию об ошибке и её положении в исходном тексте

3. Реализовать тестовую программу для демонстрации работоспособности созданного модуля
    a. Через аргументы командной строки программа должна принимать набор имён входных файлов, имя выходной директории
    b. Использовать модуль, разработанный в задании 1 для синтаксического анализа каждого входного файла и формирования набора деревьев разбора
    c. Использовать модуль, разработанный в п. 2 для формирования графов потока управления каждой подпрограммы, выявленной в синтаксической структуре текстов, содержащихся во входных файлах
    d. Для каждой обнаруженной подпрограммы вывести представление графа потока управления в отдельный файл с именем “sourceName.functionName.ext” в выходной директории, по-умолчанию размещать выходной файлы в той же директории, что соответствующий входной
    e. Для деревьев операций в графах потока управления всей совокупности подпрограмм сформировать граф вызовов, описывающий отношения между ними в плане обращения их друг к другу по именам и вывести его представление в дополнительный файл, по-умолчанию размещаемый рядом с файлом, содержащим подпрограмму main.
    f. Сообщения об ошибке должны выводиться тестовой программной (не модулем, отвечающим за анализ!) в стандартный поток вывода ошибок
4. Результаты тестирования представить в виде отчета, в который включить:
    a. В части 3 привести описание разработанных структур данных
    b. В части 4 описать программный интерфейс и особенности реализации разработанного модуля
    c. В части 5 привести примеры исходных анализируемых текстов для всех синтаксических конструкций разбираемого языка и соответствующие результаты разбора

Выполнение:

Файл заголовков, содержащий структуры использованные для создания графа потока управления:

```
#ifndef SPO_LAB1_EXECUTION_H
#define SPO_LAB1_EXECUTION_H

#include "parser.h"

typedef struct FilenameParseTree FilenameParseTree;
typedef struct ExecutionNode ExecutionNode;
typedef struct FunExecution FunExecution;
typedef struct Array Array;

struct ExecutionNode {
   char *text;
   ExecutionNode *definitely;    // безусловный переход
   ExecutionNode *conditionally; // условный переход
   TreeNode *operationTree;
   int id;
   int printed;
};

struct Array {
   int size;
   int nextPosition;
   void **elements;
};

struct FunExecution {
```

```
    char *name;
    char *filename;
    TreeNode *signature;
    TreeNode *funCalls;
    ExecutionNode *nodes;
    char **errors;
    int errorsCount;
};

struct FilenameParseTree {
    char *filename;
    ParseResult *tree;
};

Array *executionGraph(FilenameParseTree *input, int size);

void printExecution(FunExecution *funExecution, FILE *outputFunCallFile, FILE *outputOperationTreesFile,
                    FILE *outputExecutionFile);

#endif // SPO_LAB1_EXECUTION_H
```

Модуль формирует граф потока выполнения на основе дерева разбора, создаваемого модулем, созданным в первой лабораторной работе.

Помимо этого также создаются: структура вызовов функций, структура дерева операций для тех нод графа потока выполнения где это необходимо.

Парсер из первой лабораторной работы был доработан, чтобы мог корректно работать с приоритетами арифметических операций.

Модуль принимает на вход 1-ый аргумент адрес директории для выходных файлов, 2-n аргументы входных файлов.

Формирование структур происходит в файле execution.c:

```
//
// Created by Ivan on 20.10.2024.
//

#include "execution.h"
#include <string.h>

typedef struct FunCalls FunCalls;

const int START_ARRAY_SIZE = 8;
```

```
struct FunCalls {
   Array *funCalls;
   char *currentFunName;
};

Array exceptions;
int currentExecutionId = -1;

ExecutionNode *executionNode(TreeNode *treeNode, ExecutionNode *nextNode,
                        ExecutionNode *breakNode, FunCalls *funCalls);

char *mallocString(char *text) {
   char *pointer = malloc(sizeof(char) * 1024);
   sprintf(pointer, "%s", text);
   return pointer;
}

// добавление в массив с возможным динамическим расширением
void addToList(Array *currentArray, void *element) {
   void **nodes;
   if (currentArray->size != currentArray->nextPosition) {
      nodes = currentArray->elements;
   } else {
      nodes = malloc(sizeof(void *) * 2 * currentArray->size);
      for (int i = 0; i < currentArray->size; ++i) {
         nodes[i] = currentArray->elements[i];
      }
      free(currentArray->elements);
   }
   nodes[currentArray->nextPosition] = element;
   currentArray->nextPosition += 1;
}

void addException(char *text) {
   char *exception = mallocString(text);
   addToList(&exceptions, exception);
}

int getNextExecutionId() {
   currentExecutionId++;
   return currentExecutionId;
}

// утилита для получения всех node дерева разбора в виде массива (вызвано
// бинарной реализацией листов)
Array findListItemsUtil(TreeNode *treeNode) {
   TreeNode **nodes = malloc(sizeof(TreeNode *) * START_ARRAY_SIZE);
   Array items = {START_ARRAY_SIZE, 0, nodes};

   TreeNode *currentListNode = treeNode;
   do {
      if (currentListNode->childrenNumber == 0) {
         currentListNode = NULL;
      } else if (currentListNode->childrenNumber == 1) {
         addToList(&items, currentListNode->childNodes[0]);
         currentListNode = NULL;
```

```
      } else if (currentListNode->childrenNumber == 2) {
         addToList(&items, currentListNode->childNodes[0]);
         currentListNode = currentListNode->childNodes[1];
      } else {
         char exceptionText[1024];
         sprintf(exceptionText,
               "Exception in list parsing more than two by element id %d",
               currentListNode->id);
         addException(exceptionText);
         return items;
      }
   } while (currentListNode != NULL);
   return items;
}

// получение корневого элемента из результатов парсинга
TreeNode *findSourceNode(FilenameParseTree input) {
   TreeNode **inputNodes = input.tree->nodes;
   int inputNodesSize = input.tree->size;
   TreeNode *sourceNode = inputNodes[inputNodesSize - 1];
   return sourceNode;
}

// получение списка функций из корневого элемента
Array findSourceItems(TreeNode *source) {
   if (source->childrenNumber != 0) {
      TreeNode *sourceItemsList = source->childNodes[0];
      return findListItemsUtil(sourceItemsList);
   } else {
      return (Array) {0, 0, NULL};
   }
}

ExecutionNode *initExecutionNode(char *text) {
   ExecutionNode *node = malloc(sizeof(ExecutionNode));
   node->id = getNextExecutionId();
   node->text = mallocString(text);
   node->definitely = NULL;
   node->conditionally = NULL;
   node->operationTree = NULL;
   node->printed = 0;
   return node;
}

TreeNode *mallocTreeNode(char *type, char *value, int nodeNumber) {
   TreeNode *node = malloc(sizeof(TreeNode));
   node->id = getNextExecutionId();
   if (type) {
      node->type = mallocString(type);
   } else {
      node->type = NULL;
   }
   if (value) {
      node->value = mallocString(value);
   } else {
      node->value = NULL;
```

```
    }
    node->childNodes = malloc(sizeof(TreeNode *) * nodeNumber);
    node->childrenNumber = nodeNumber;
    return node;
}

// создание блока listStatement
ExecutionNode *executionListStatementNode(TreeNode *treeNode,
                                          ExecutionNode *nextNode,
                                          ExecutionNode *breakNode,
                                          FunCalls *funCalls) {
    ExecutionNode *tmpNextNode = nextNode;
    if (treeNode->childrenNumber == 2) {
        tmpNextNode = executionNode(treeNode->childNodes[1], nextNode, breakNode, funCalls);
    }
    ExecutionNode *node = initExecutionNode("");
    node->definitely =
            executionNode(treeNode->childNodes[0], tmpNextNode, breakNode, funCalls);
    return node;
}

ExecutionNode *executionVarNode(TreeNode *treeNode, ExecutionNode *nextNode,
                                ExecutionNode *breakNode) {
    ExecutionNode *node = initExecutionNode("");
    Array variablesList = {0, 0, NULL};
    TreeNode *typeNode = NULL;
    if (treeNode->childrenNumber == 2) {
        typeNode = treeNode->childNodes[1];
        variablesList = findListItemsUtil(treeNode->childNodes[0]);
    } else {
        typeNode = treeNode->childNodes[0];
    }
    char resultNodeType[1024];
    if (!strcmp(typeNode->type, "array")) {
        int size = 0;
        if (typeNode->childrenNumber == 2) {
            size = findListItemsUtil(typeNode->childNodes[1]).nextPosition;
        }
        sprintf(resultNodeType,
                "array of %s size %d",
                typeNode->value, size);
    } else {
        sprintf(resultNodeType, "%s", typeNode->value);
    }

    ExecutionNode *previous = node;
    for (int i = 0; i < variablesList.nextPosition; ++i) {
        char varNameAndType[1024];
        sprintf(varNameAndType,
                "%s as %s",
                ((TreeNode *) variablesList.elements[i])->value, resultNodeType);
        previous->definitely = initExecutionNode(varNameAndType);
        previous = previous->definitely;
    }
    previous->definitely = nextNode;
    return node;
```

```
}

//чтобы распарсить дерево операций в последовательность операций
/*ExecutionNode *executionExpressionNode(TreeNode *treeNode, ExecutionNode *nextNode,
ExecutionNode *conditionallyNext,
                            ExecutionNode *parentNode) {
   ExecutionNode *node = initExecutionNode("");
   int hasNewParent = 0;
   if (parentNode == NULL) {
      hasNewParent = 1;
      parentNode = initExecutionNode("");
   }
   node->definitely = nextNode;
   node->conditionally = conditionallyNext;
   if (treeNode->childrenNumber == 2) {
      ExecutionNode *leftNode = executionExpressionNode(treeNode->childNodes[0], NULL, NULL,
parentNode);
      ExecutionNode *rightNode = executionExpressionNode(treeNode->childNodes[1], node, NULL,
leftNode);
   } else if (treeNode->childrenNumber != 0 && !strcmp(treeNode->childNodes[0]->type, "braces")) {
      executionExpressionNode(treeNode->childNodes[0], node, NULL, parentNode);
   } else {
      parentNode->definitely = node;
   }
   if (hasNewParent) {
      return parentNode;
   } else {
      return node;
   }
}*/

// для построения дерева операций
TreeNode *operationTreeNode(TreeNode *parsingTree, FunCalls *funCalls) {
   TreeNode *node = NULL;

   if (!strcmp(parsingTree->type, "braces")) {
      return operationTreeNode(parsingTree->childNodes[0], funCalls);
   } else if (!strcmp(parsingTree->type, "INCREMENT") || !strcmp(parsingTree->type, "DECREMENT")) {
      node = mallocTreeNode("SET", NULL, 2);
      char valuePlace[1024];
      sprintf(valuePlace,
            "value place '%s'",
            parsingTree->childNodes[0]->value);
      node->childNodes[0] = mallocTreeNode(NULL, valuePlace, 0);
      if (!strcmp(parsingTree->type, "INCREMENT")) {
         node->childNodes[1] = mallocTreeNode("PLUS", NULL, 2);
      } else {
         node->childNodes[1] = mallocTreeNode("MINUS", NULL, 2);
      }
      node->childNodes[1]->childNodes[0] = mallocTreeNode(NULL, "const: 1", 0);
      node->childNodes[1]->childNodes[1] = operationTreeNode(parsingTree->childNodes[0], funCalls);
   } else if (!strcmp(parsingTree->type, "callOrIndexer")) {
      char executionName[1024];
      sprintf(executionName,
            "execute: %s",
            parsingTree->childNodes[0]->value);
```

```
      if (parsingTree->childrenNumber == 1) {
         node = mallocTreeNode(NULL, executionName, 0);
      } else {
         Array argsArray = findListItemsUtil(parsingTree->childNodes[1]);
         node = mallocTreeNode(NULL, executionName, argsArray.nextPosition);
         for (int i = 0; i < argsArray.nextPosition; ++i) {
            node->childNodes[i] = operationTreeNode(argsArray.elements[i], funCalls);
         }
      }
      char funCallOperationIdNodeString[1024];
      sprintf(funCallOperationIdNodeString, "%d", node->id);
      TreeNode *funCallOperationIdNode = mallocTreeNode("operationTreeId",
funCallOperationIdNodeString, 1);
      TreeNode *calledFunNameNode = mallocTreeNode("call", parsingTree->childNodes[0]->value, 0);
      funCallOperationIdNode->childNodes[0] = calledFunNameNode;
      addToList(funCalls->funCalls, funCallOperationIdNode);
   } else if (parsingTree->childrenNumber == 2) {
      node = mallocTreeNode(parsingTree->type, parsingTree->value, parsingTree->childrenNumber);

      if (!strcmp(parsingTree->type, "SET")) {
         char valuePlace[1024];
         sprintf(valuePlace,
               "value place '%s'",
               parsingTree->childNodes[0]->value);
         node->childNodes[0] = mallocTreeNode(NULL, valuePlace, 0);
         node->childNodes[1] = operationTreeNode(parsingTree->childNodes[1], funCalls);
      } else {
         node->childNodes[0] = operationTreeNode(parsingTree->childNodes[0], funCalls);
         node->childNodes[1] = operationTreeNode(parsingTree->childNodes[1], funCalls);
      }
   } else if (parsingTree->childrenNumber == 0) {
      if (!strcmp(parsingTree->type, "IDENTIFIER")) {
         node = mallocTreeNode("read", NULL, 1);
         char valuePlace[1024];
         sprintf(valuePlace,
               "value place '%s'",
               parsingTree->value);
         node->childNodes[0] = mallocTreeNode(NULL, valuePlace, 0);
      } else {
         char constVal[1024];
         sprintf(constVal,
               "const: %s",
               parsingTree->value);
         node = mallocTreeNode(NULL, constVal, 0);
      }
   }
   return node;
}

char *expressionNodeToString(TreeNode *treeNode) {
   // TODO("делать доп действия и создавать доп узлы для каждого конкретного типа node")
   if (treeNode->childrenNumber == 0) {
      return mallocString(treeNode->value);
   } else if (treeNode->childrenNumber == 1) {
      char *childStr = expressionNodeToString(treeNode->childNodes[0]);
      char text[1024];
```

```
    sprintf(text,
        "%s-%s-",
        treeNode->type, childStr);
    return mallocString(text);
  } else {
    char *childLeftStr = expressionNodeToString(treeNode->childNodes[0]);
    char *childRightStr = expressionNodeToString(treeNode->childNodes[1]);
    char text[1024];
    sprintf(text,
        "%s %s %s",
        childLeftStr, treeNode->type, childRightStr);
    return mallocString(text);
  }
}

ExecutionNode *executionExpressionNode(TreeNode *treeNode, FunCalls *funCalls) {
  ExecutionNode *node = initExecutionNode(expressionNodeToString(treeNode));
  node->operationTree = operationTreeNode(treeNode, funCalls);
  return node;
}

ExecutionNode *executionElseNode(TreeNode *treeNode, ExecutionNode *nextNode,
                    ExecutionNode *breakNode, FunCalls *funCalls) {
  ExecutionNode *node = initExecutionNode("");
  if (treeNode->childrenNumber == 1) {
    node->definitely = executionNode(treeNode->childNodes[0],
                        nextNode, breakNode, funCalls);
  } else {
    node->definitely = nextNode;
  }
  return node;
}

ExecutionNode *executionIfNode(TreeNode *treeNode, ExecutionNode *nextNode,
                    ExecutionNode *breakNode, FunCalls *funCalls) {
  ExecutionNode *node = initExecutionNode("");
  TreeNode *elseTreeNode = NULL;
  TreeNode *ifStatements = NULL;
  if (treeNode->childrenNumber == 3) {
    //  случай когда есть стейтменты в if и существует else
    ifStatements = treeNode->childNodes[1];
    elseTreeNode = treeNode->childNodes[2];
  } else if (treeNode->childrenNumber == 2 &&
        !strcmp(treeNode->childNodes[1]->type, "else")) {
    //  когда нет стейтментов в if
    elseTreeNode = treeNode->childNodes[1];
  } else if (treeNode->childrenNumber == 2) {
    ifStatements = treeNode->childNodes[1];
  }

  ExecutionNode *conditionNextNode = NULL;
  ExecutionNode *conditionConditionallyNode = NULL;
  if (elseTreeNode != NULL) {
    ExecutionNode *elseNode =
        executionElseNode(elseTreeNode, nextNode, breakNode, funCalls);
    conditionNextNode = elseNode;
```

```
    } else {
        conditionNextNode = nextNode;
    }
    if (ifStatements != NULL) {
        ExecutionNode *statementsNode =
                executionNode(ifStatements, nextNode, breakNode, funCalls);
        conditionConditionallyNode = statementsNode;
    }
    ExecutionNode *conditionNode = executionExpressionNode(treeNode->childNodes[0], funCalls);
    node->definitely = conditionNode;
    conditionNode->definitely = conditionNextNode;
    conditionNode->conditionally = conditionConditionallyNode;
    return node;
}

ExecutionNode *executionWhileNode(TreeNode *treeNode, ExecutionNode *nextNode,
                                  ExecutionNode *breakNode, FunCalls *funCalls) {
    ExecutionNode *node = initExecutionNode("");
    ExecutionNode *statementNode = NULL;
    if (treeNode->childrenNumber == 2) {
        statementNode = executionNode(treeNode->childNodes[1], node, nextNode, funCalls);
    } else {
        statementNode = initExecutionNode("");
        statementNode->definitely = node;
    }
    ExecutionNode *conditionNode = executionExpressionNode(treeNode->childNodes[0], funCalls);
    node->definitely = conditionNode;
    conditionNode->definitely = nextNode;
    conditionNode->conditionally = statementNode;
    return node;
}

ExecutionNode *executionDoNode(TreeNode *treeNode, ExecutionNode *nextNode,
                               ExecutionNode *breakNode, FunCalls *funCalls) {
    ExecutionNode *node = initExecutionNode("");
    TreeNode *conditionTreeNode = NULL;
    if (treeNode->childrenNumber == 3) {
        conditionTreeNode = treeNode->childNodes[2];
    } else {
        conditionTreeNode = treeNode->childNodes[1];
    }
    ExecutionNode *doConditionNode = executionExpressionNode(conditionTreeNode, funCalls);
    doConditionNode->definitely = nextNode;
    doConditionNode->conditionally = node;
    ExecutionNode *statementNode = NULL;
    if (treeNode->childrenNumber == 3) {
        statementNode = executionNode(treeNode->childNodes[0], doConditionNode, nextNode, funCalls);
    } else {
        statementNode = initExecutionNode("");
        statementNode->definitely = doConditionNode;
    }
    node->definitely = statementNode;
    return node;
}

ExecutionNode *executionBreakNode(TreeNode *treeNode, ExecutionNode *nextNode,
```

```
                        ExecutionNode *breakNode) {
   ExecutionNode *node = initExecutionNode("");
   if (breakNode == NULL) {
      char exceptionText[1024];
      sprintf(exceptionText,
            "Exception in BREAK tree node parsing id --> %d no loop for break found",
            treeNode[0].id);
      addException(exceptionText);
      ExecutionNode *exceptionNode = initExecutionNode(exceptionText);
      node->definitely = exceptionNode;
      exceptionNode->definitely = nextNode;
   } else {
      node->definitely = breakNode;
   }
   return node;
}

// созадние блока
ExecutionNode *executionNode(TreeNode *treeNode, ExecutionNode *nextNode,
                     ExecutionNode *breakNode, FunCalls *funCalls) {
   if (!strcmp(treeNode[0].type, "listStatement")) {
      return executionListStatementNode(treeNode, nextNode, breakNode, funCalls);
   } else if (!strcmp(treeNode[0].type, "var")) {
      return executionVarNode(treeNode, nextNode, breakNode);
   } else if (!strcmp(treeNode[0].type, "if")) {
      return executionIfNode(treeNode, nextNode, breakNode, funCalls);
   } else if (!strcmp(treeNode[0].type, "while")) {
      return executionWhileNode(treeNode, nextNode, breakNode, funCalls);
   } else if (!strcmp(treeNode[0].type, "break")) {
      return executionBreakNode(treeNode, nextNode, breakNode);
   } else if (!strcmp(treeNode[0].type, "do")) {
      return executionDoNode(treeNode, nextNode, breakNode, funCalls);
   } else {
//       expression
      ExecutionNode *expressionNode = executionExpressionNode(treeNode, funCalls);
      expressionNode->definitely = nextNode;
   }
}

ExecutionNode *initGraph(TreeNode *sourceItem, FunCalls *funCalls) {
   ExecutionNode *startNode = initExecutionNode("START");
   ExecutionNode *endNode = initExecutionNode("FINISH");
   TreeNode *funcDef = sourceItem->childNodes[0];
   if (funcDef->childrenNumber == 2) {
      ExecutionNode *listStatementNode =
            executionNode(funcDef->childNodes[1], endNode, NULL, funCalls);
      startNode->definitely = listStatementNode;
   } else {
      startNode->definitely = endNode;
   }
   return startNode;
}

void initExceptions() {
   void **nodes = malloc(sizeof(char *) * START_ARRAY_SIZE);
   exceptions = (Array) {START_ARRAY_SIZE, 0, nodes};
```

```
}

Array *executionGraph(FilenameParseTree *input, int size) {
   void **resultNodes = malloc(sizeof(FunExecution *) * START_ARRAY_SIZE);
   Array *result = malloc(sizeof(Array));
   result->size = START_ARRAY_SIZE;
   result->nextPosition = 0;
   result->elements = resultNodes;

   for (int i = 0; i < size; ++i) {
      initExceptions();
      TreeNode *sourceNode = findSourceNode(input[i]);
      Array sourceItems = findSourceItems(sourceNode);
      for (int j = 0; j < sourceItems.nextPosition; ++j) {
         FunExecution *currentFunExecution = malloc(sizeof(FunExecution));
         currentFunExecution->filename = input[i].filename;
         TreeNode **currentSourceItemElements =
               ((TreeNode **) sourceItems.elements);
         currentFunExecution->name =
               currentSourceItemElements[j]->childNodes[0]->childNodes[0]->value;
         currentFunExecution->signature =
               currentSourceItemElements[j]->childNodes[0];

         void **nodes = malloc(sizeof(TreeNode *) * START_ARRAY_SIZE);
         Array funs = (Array) {START_ARRAY_SIZE, 0, nodes};
         FunCalls funCalls = (FunCalls) {&funs, currentFunExecution->name};
         currentFunExecution->nodes = initGraph(currentSourceItemElements[j], &funCalls);
         TreeNode *funCallsRoot = mallocTreeNode("currentFunction", currentFunExecution->name,
                                    funCalls.funCalls->nextPosition);
         for (int k = 0; k < funCalls.funCalls->nextPosition; ++k) {
            funCallsRoot->childNodes[k] = funCalls.funCalls->elements[k];
         }
         currentFunExecution->funCalls = funCallsRoot;
         currentFunExecution->errorsCount = exceptions.nextPosition;
         currentFunExecution->errors = exceptions.elements;
         addToList(result, currentFunExecution);
      }
   }
   return result;
}

void printNode(TreeNode *node, FILE *outputFile) {
   if (node == NULL) {
      return;
   }
   int childrenNumber = node->childrenNumber;
   for (int i = 0; i < childrenNumber; ++i) {
      printNode(node->childNodes[i], outputFile);
      fprintf(outputFile, "node%d", node->id);
      fprintf(outputFile, "([");
      int typeExists = 0;
      if (node->type != NULL && strlen(node->type) > 0) {
         fprintf(outputFile, "Type: %s", node->type);
         typeExists = 1;
      }
      if (node->value != NULL && strlen(node->value) > 0) {
```

```
            if (typeExists) {
                fprintf(outputFile, ", ", node->value);
            }
            fprintf(outputFile, "Value: %s", node->value);
        }
        fprintf(outputFile, "])");

        TreeNode *childNode = node->childNodes[i];
        fprintf(outputFile, " --> ");
        fprintf(outputFile, "node%d", childNode->id);
        fprintf(outputFile, "([");
        typeExists = 0;
        if (childNode->type != NULL && strlen(childNode->type) > 0) {
            fprintf(outputFile, "Type: %s", childNode->type);
            typeExists = 1;
        }
        if (childNode->value != NULL && strlen(childNode->value) > 0) {
            if (typeExists) {
                fprintf(outputFile, ", ");
            }
            fprintf(outputFile, "Value: %s", childNode->value);
        }
        fprintf(outputFile, "])");
        fprintf(outputFile, "\n");
    }
}

void printTreeNode(TreeNode *node, FILE *outputFile) {
    fprintf(outputFile, "flowchart TB\n");
    printNode(node, outputFile);
    fprintf(outputFile, "\n");
}

void printExecutionNode(ExecutionNode *father, ExecutionNode *child, FILE *outputFile, char
*relationName) {
    fprintf(outputFile, "node%d", father->id);
    fprintf(outputFile, "([");
    fprintf(outputFile, "Text: %s", father->text);
    fprintf(outputFile, "])");
    fprintf(outputFile, " --%s--> ", relationName);
    fprintf(outputFile, "node%d", child->id);
    fprintf(outputFile, "([");
    fprintf(outputFile, "Text: %s", child->text);
    fprintf(outputFile, "])");
    fprintf(outputFile, "\n");
}

void printExecutionGraphNodeToFile(ExecutionNode *executionNode, FILE *outputOperationTreesFile,
                                   FILE *outputExecutionFile) {
    if (executionNode->printed) {
        return;
    } else {
        executionNode->printed = 1;
    }

    if (executionNode->operationTree) {
```

```
      char linkedExecutionNodeId[1024];
      sprintf(linkedExecutionNodeId, "%d", executionNode->id);
      TreeNode *linkedExecutionNode = mallocTreeNode("linked execution node id", linkedExecutionNodeId,
1);
      linkedExecutionNode->childNodes[0] = executionNode->operationTree;
      printNode(linkedExecutionNode, outputOperationTreesFile);
   }

   ExecutionNode *definitely = executionNode->definitely;
   if (definitely) {
      printExecutionGraphNodeToFile(definitely, outputOperationTreesFile, outputExecutionFile);

      printExecutionNode(executionNode, definitely, outputExecutionFile, "definitely");
   }

   ExecutionNode *conditionally = executionNode->conditionally;
   if (conditionally) {
      printExecutionGraphNodeToFile(conditionally, outputOperationTreesFile, outputExecutionFile);

      printExecutionNode(executionNode, conditionally, outputExecutionFile, "conditionally");
   }
}

void printExecutionGraphToFile(ExecutionNode *executionNode, FILE *outputOperationTreesFile,
                        FILE *outputExecutionFile) {
   fprintf(outputOperationTreesFile, "flowchart TB\n");
   fprintf(outputExecutionFile, "flowchart TB\n");
   printExecutionGraphNodeToFile(executionNode, outputOperationTreesFile, outputExecutionFile);
   fprintf(outputOperationTreesFile, "\n");
   fprintf(outputExecutionFile, "\n");
}

void printExecution(FunExecution *funExecution, FILE *outputFunCallFile, FILE *outputOperationTreesFile,
               FILE *outputExecutionFile) {
   printTreeNode(funExecution->funCalls, outputFunCallFile);
   printExecutionGraphToFile(funExecution->nodes, outputOperationTreesFile, outputExecutionFile);
}
```

Результаты:

Входной файл:

```
function main3(arg1 as bool, arg2 as int, arg3 as bool) as bool(,,,)
   ++a;
   dim bbb as asd
   1+1*fun(bbb, ccc);
   dim a,b,c as uint()
   dim d as string
   dim as bool
   a = 10;
   (++a * 1 + fun()) + 1 == 10;
   do
      1+1;
   loop while (1+1)-1+(1+2) == 2
   do
   loop while 3 == 2
   do
      break
   loop until 2+2 == 4
   if 1+(1-1) == 1 then
      if 1 == 2 then
         while (1+1 == 2)
            break
            a = 1+1;
         wend

      else
         do loop while 1>2
         do
            dim a as int
            a = b+c;
         loop until 2 == 2
         "str";
         (a+b) %c;
         asd; //just identifer and comment test
      end if
   end if
   if 1 == 1 then
   end if
end function

function main(arg1 as bool, arg2 as int, arg3 as bool) as bool(,,,)
   dim bbb as asd
   1+1*fun(bbb);
   dim a,b,c as uint()
   dim d as string
   dim as bool
   a = 10;
   (++a * 1 + fun()) + 1 == 10;
   do
      1+1;
   loop while (1+1)-1+(1+2) == 2
   do
```

```
  loop while 3 == 2
  do
    break
  loop until 2+2 == 4
  if 1+(1-1) == 1 then
    if 1 == 2 then
      while (1+1 == 2)
        break
        a = 1+1;
      wend

    else
      do loop while 1>2
      do
        dim a as int
        a = b+c;
      loop until 2 == 2
      "str";
      (a+b) %c;
      asd; //just identifer and comment test
    end if
  end if
  if 1 == 1 then
  end if
end function
```

В результате работы модуля получаются следующие структуры:

- Пример графа потока выполнения:

```
flowchart TB
node524([Text: true]) --definitely--> node393([Text: FINISH])
node397([Text: print_num callOrIndexer listExpr-res-]) --definitely--> node395([Text: ])
node396([Text: ]) --definitely--> node397([Text: print_num callOrIndexer listExpr-res-])
node418([Text: op EQUALITY /]) --definitely--> node396([Text: ])
node407([Text: res SET div callOrIndexer a listExpr listExpr-b-]) --definitely--> node396([Text: ])
node406([Text: ]) --definitely--> node407([Text: res SET div callOrIndexer a listExpr listExpr-b-])
node418([Text: op EQUALITY /]) --conditionally--> node406([Text: ])
node405([Text: ]) --definitely--> node418([Text: op EQUALITY /])
node404([Text: ]) --definitely--> node405([Text: ])
node439([Text: op EQUALITY *]) --definitely--> node404([Text: ])
node428([Text: res SET mul callOrIndexer a listExpr listExpr-b-]) --definitely--> node404([Text: ])
node427([Text: ]) --definitely--> node428([Text: res SET mul callOrIndexer a listExpr listExpr-b-])
node439([Text: op EQUALITY *]) --conditionally--> node427([Text: ])
node426([Text: ]) --definitely--> node439([Text: op EQUALITY *])
node425([Text: ]) --definitely--> node426([Text: ])
node460([Text: op EQUALITY -]) --definitely--> node425([Text: ])
node449([Text: res SET sub callOrIndexer a listExpr listExpr-b-]) --definitely--> node425([Text: ])
node448([Text: ]) --definitely--> node449([Text: res SET sub callOrIndexer a listExpr listExpr-b-])
node460([Text: op EQUALITY -]) --conditionally--> node448([Text: ])
```

```
node447([Text: ]) --definitely--> node460([Text: op EQUALITY -])
node446([Text: ]) --definitely--> node447([Text: ])
node481([Text: op EQUALITY +]) --definitely--> node446([Text: ])
node470([Text: res SET sum callOrIndexer a listExpr listExpr-b-]) --definitely--> node446([Text: ])
node469([Text: ]) --definitely--> node470([Text: res SET sum callOrIndexer a listExpr listExpr-b-])
node481([Text: op EQUALITY +]) --conditionally--> node469([Text: ])
node468([Text: ]) --definitely--> node481([Text: op EQUALITY +])
node467([Text: ]) --definitely--> node468([Text: ])
node489([Text: b SET callOrIndexer-read_num-]) --definitely--> node467([Text: ])
node488([Text: ]) --definitely--> node489([Text: b SET callOrIndexer-read_num-])
node497([Text: callOrIndexer-stdin-]) --definitely--> node488([Text: ])
node496([Text: ]) --definitely--> node497([Text: callOrIndexer-stdin-])
node503([Text: callOrIndexer-stdin-]) --definitely--> node496([Text: ])
node502([Text: ]) --definitely--> node503([Text: callOrIndexer-stdin-])
node509([Text: op SET callOrIndexer-stdin-]) --definitely--> node502([Text: ])
node508([Text: ]) --definitely--> node509([Text: op SET callOrIndexer-stdin-])
node517([Text: a SET callOrIndexer-read_num-]) --definitely--> node508([Text: ])
node516([Text: ]) --definitely--> node517([Text: a SET callOrIndexer-read_num-])
node524([Text: true]) --conditionally--> node516([Text: ])
node395([Text: ]) --definitely--> node524([Text: true])
node394([Text: ]) --definitely--> node395([Text: ])
node530([Text: AS res int]) --definitely--> node394([Text: ])
node529([Text: ]) --definitely--> node530([Text: AS res int])
node528([Text: ]) --definitely--> node529([Text: ])
node536([Text: AS op int]) --definitely--> node528([Text: ])
node535([Text: ]) --definitely--> node536([Text: AS op int])
node534([Text: ]) --definitely--> node535([Text: ])
node542([Text: AS b int]) --definitely--> node534([Text: ])
node541([Text: ]) --definitely--> node542([Text: AS b int])
node540([Text: ]) --definitely--> node541([Text: ])
node548([Text: AS a int]) --definitely--> node540([Text: ])
node547([Text: ]) --definitely--> node548([Text: AS a int])
node546([Text: ]) --definitely--> node547([Text: ])
node552([Text: ]) --definitely--> node546([Text: ])
node392([Text: START]) --definitely--> node552([Text: ])
```

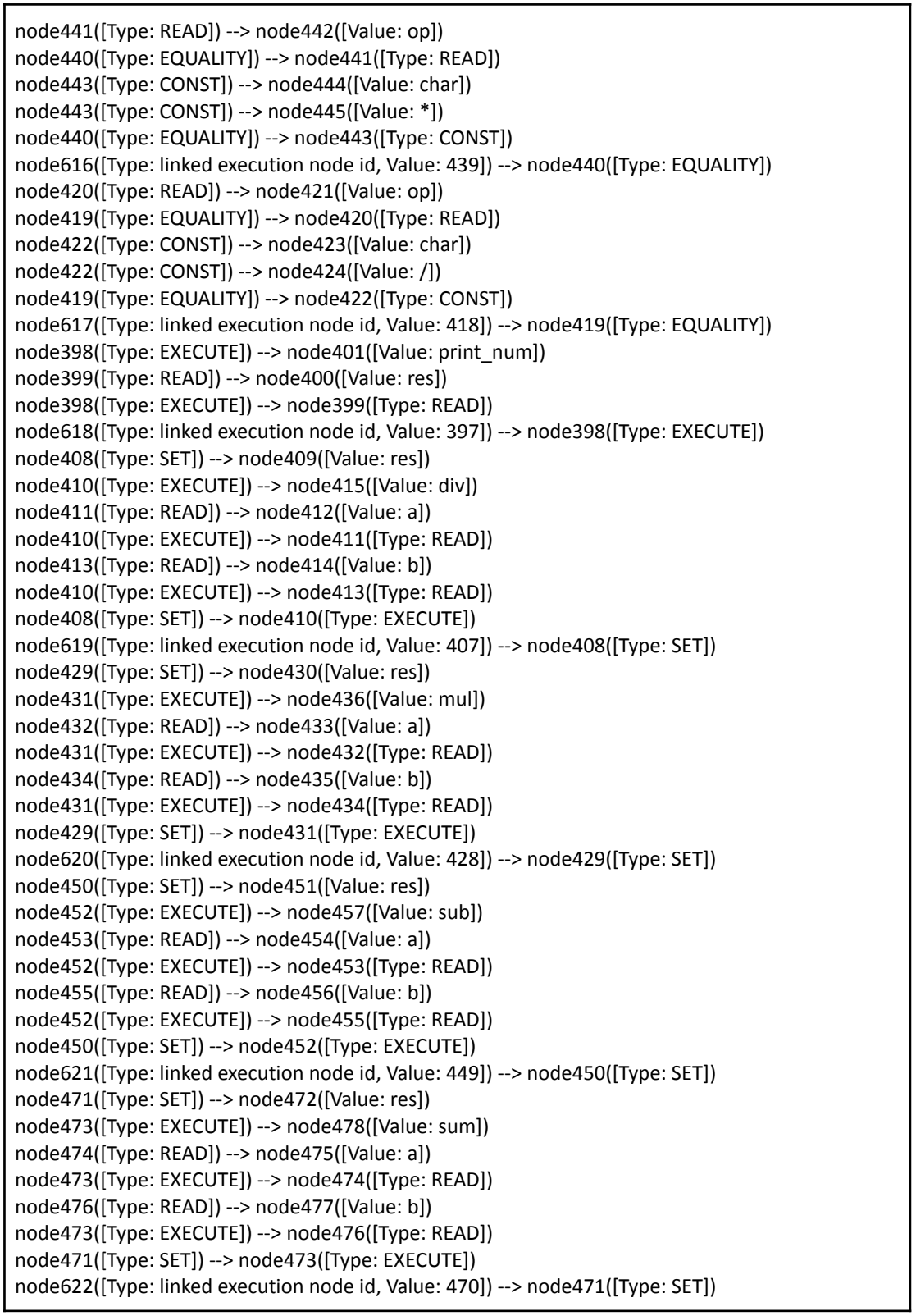
- Пример структуры вызова функций:

```
flowchart TB
node402([Type: operationTreeId, Value: 398]) --> node403([Type: call, Value: print_num])
node553([Type: currentFunction, Value: main]) --> node402([Type: operationTreeId, Value: 398])
node416([Type: operationTreeId, Value: 410]) --> node417([Type: call, Value: div])
node553([Type: currentFunction, Value: main]) --> node416([Type: operationTreeId, Value: 410])
node437([Type: operationTreeId, Value: 431]) --> node438([Type: call, Value: mul])
node553([Type: currentFunction, Value: main]) --> node437([Type: operationTreeId, Value: 431])
node458([Type: operationTreeId, Value: 452]) --> node459([Type: call, Value: sub])
node553([Type: currentFunction, Value: main]) --> node458([Type: operationTreeId, Value: 452])
node479([Type: operationTreeId, Value: 473]) --> node480([Type: call, Value: sum])
node553([Type: currentFunction, Value: main]) --> node479([Type: operationTreeId, Value: 473])
node494([Type: operationTreeId, Value: 492]) --> node495([Type: call, Value: read_num])
node553([Type: currentFunction, Value: main]) --> node494([Type: operationTreeId, Value: 492])
node500([Type: operationTreeId, Value: 498]) --> node501([Type: call, Value: stdin])
node553([Type: currentFunction, Value: main]) --> node500([Type: operationTreeId, Value: 498])
```

```
node506([Type: operationTreeId, Value: 504]) --> node507([Type: call, Value: stdin])
node553([Type: currentFunction, Value: main]) --> node506([Type: operationTreeId, Value: 504])
node514([Type: operationTreeId, Value: 512]) --> node515([Type: call, Value: stdin])
node553([Type: currentFunction, Value: main]) --> node514([Type: operationTreeId, Value: 512])
node522([Type: operationTreeId, Value: 520]) --> node523([Type: call, Value: read_num])
node553([Type: currentFunction, Value: main]) --> node522([Type: operationTreeId, Value: 520])
```

- Пример структуры дерева операций:

```
flowchart TB
node549([Type: AS]) --> node550([Value: int])
node549([Type: AS]) --> node551([Value: a])
node604([Type: linked execution node id, Value: 548]) --> node549([Type: AS])
node543([Type: AS]) --> node544([Value: int])
node543([Type: AS]) --> node545([Value: b])
node605([Type: linked execution node id, Value: 542]) --> node543([Type: AS])
node537([Type: AS]) --> node538([Value: int])
node537([Type: AS]) --> node539([Value: op])
node606([Type: linked execution node id, Value: 536]) --> node537([Type: AS])
node531([Type: AS]) --> node532([Value: int])
node531([Type: AS]) --> node533([Value: res])
node607([Type: linked execution node id, Value: 530]) --> node531([Type: AS])
node525([Type: CONST]) --> node526([Value: bool])
node525([Type: CONST]) --> node527([Value: true])
node608([Type: linked execution node id, Value: 524]) --> node525([Type: CONST])
node518([Type: SET]) --> node519([Value: a])
node520([Type: EXECUTE]) --> node521([Value: read_num])
node518([Type: SET]) --> node520([Type: EXECUTE])
node609([Type: linked execution node id, Value: 517]) --> node518([Type: SET])
node510([Type: SET]) --> node511([Value: op])
node512([Type: EXECUTE]) --> node513([Value: stdin])
node510([Type: SET]) --> node512([Type: EXECUTE])
node610([Type: linked execution node id, Value: 509]) --> node510([Type: SET])
node504([Type: EXECUTE]) --> node505([Value: stdin])
node611([Type: linked execution node id, Value: 503]) --> node504([Type: EXECUTE])
node498([Type: EXECUTE]) --> node499([Value: stdin])
node612([Type: linked execution node id, Value: 497]) --> node498([Type: EXECUTE])
node490([Type: SET]) --> node491([Value: b])
node492([Type: EXECUTE]) --> node493([Value: read_num])
node490([Type: SET]) --> node492([Type: EXECUTE])
node613([Type: linked execution node id, Value: 489]) --> node490([Type: SET])
node483([Type: READ]) --> node484([Value: op])
node482([Type: EQUALITY]) --> node483([Type: READ])
node485([Type: CONST]) --> node486([Value: char])
node485([Type: CONST]) --> node487([Value: +])
node482([Type: EQUALITY]) --> node485([Type: CONST])
node614([Type: linked execution node id, Value: 481]) --> node482([Type: EQUALITY])
node462([Type: READ]) --> node463([Value: op])
node461([Type: EQUALITY]) --> node462([Type: READ])
node464([Type: CONST]) --> node465([Value: char])
node464([Type: CONST]) --> node466([Value: -])
node461([Type: EQUALITY]) --> node464([Type: CONST])
node615([Type: linked execution node id, Value: 460]) --> node461([Type: EQUALITY])
```


node441([Type: READ]) --> node442([Value: op])
node440([Type: EQUALITY]) --> node441([Type: READ])
node443([Type: CONST]) --> node444([Value: char])
node443([Type: CONST]) --> node445([Value: *])
node440([Type: EQUALITY]) --> node443([Type: CONST])
node616([Type: linked execution node id, Value: 439]) --> node440([Type: EQUALITY])
node420([Type: READ]) --> node421([Value: op])
node419([Type: EQUALITY]) --> node420([Type: READ])
node422([Type: CONST]) --> node423([Value: char])
node422([Type: CONST]) --> node424([Value: /])
node419([Type: EQUALITY]) --> node422([Type: CONST])
node617([Type: linked execution node id, Value: 418]) --> node419([Type: EQUALITY])
node398([Type: EXECUTE]) --> node401([Value: print_num])
node399([Type: READ]) --> node400([Value: res])
node398([Type: EXECUTE]) --> node399([Type: READ])
node618([Type: linked execution node id, Value: 397]) --> node398([Type: EXECUTE])
node408([Type: SET]) --> node409([Value: res])
node410([Type: EXECUTE]) --> node415([Value: div])
node411([Type: READ]) --> node412([Value: a])
node410([Type: EXECUTE]) --> node411([Type: READ])
node413([Type: READ]) --> node414([Value: b])
node410([Type: EXECUTE]) --> node413([Type: READ])
node408([Type: SET]) --> node410([Type: EXECUTE])
node619([Type: linked execution node id, Value: 407]) --> node408([Type: SET])
node429([Type: SET]) --> node430([Value: res])
node431([Type: EXECUTE]) --> node436([Value: mul])
node432([Type: READ]) --> node433([Value: a])
node431([Type: EXECUTE]) --> node432([Type: READ])
node434([Type: READ]) --> node435([Value: b])
node431([Type: EXECUTE]) --> node434([Type: READ])
node429([Type: SET]) --> node431([Type: EXECUTE])
node620([Type: linked execution node id, Value: 428]) --> node429([Type: SET])
node450([Type: SET]) --> node451([Value: res])
node452([Type: EXECUTE]) --> node457([Value: sub])
node453([Type: READ]) --> node454([Value: a])
node452([Type: EXECUTE]) --> node453([Type: READ])
node455([Type: READ]) --> node456([Value: b])
node452([Type: EXECUTE]) --> node455([Type: READ])
node450([Type: SET]) --> node452([Type: EXECUTE])
node621([Type: linked execution node id, Value: 449]) --> node450([Type: SET])
node471([Type: SET]) --> node472([Value: res])
node473([Type: EXECUTE]) --> node478([Value: sum])
node474([Type: READ]) --> node475([Value: a])
node473([Type: EXECUTE]) --> node474([Type: READ])
node476([Type: READ]) --> node477([Value: b])
node473([Type: EXECUTE]) --> node476([Type: READ])
node471([Type: SET]) --> node473([Type: EXECUTE])
node622([Type: linked execution node id, Value: 470]) --> node471([Type: SET])

Вывод:

В результате выполнения лабораторной работы познакомился с принципами построения графа потока управления. Научился строить дерево операций. Узнал как создавать структуру вызова функциями друг друга.