Университет ИТМО

Факультет ПИиКТ

Системное программное обеспечение

Лабораторная работа №2

Работу выполнил:

Абузов Ярослав

Группа:

P4114

Вариант:

1

Преподаватель:

Кореньков Ю. Д.

Санкт-Петербург

2024

Содержание

1 Цели 3

2 Задачи 3

3 Описание работы 5

3.1 Описание тестовой программы 5

3.2 Описание модуля для построения и работы с CFG 8

3.3 Описание модуля для построения Operation Tree 15

3.4 Модуль для работы с Call Graph 18

4 Примеры 20

5 Выводы 28

1 Цели

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.

2 Задачи

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

3 Описание работы

3.1 Описание тестовой программы

Аргументы командной строки:

- `-d` или `--debug`: включает режим отладки, при котором выводится дополнительная диагностическая информация.
- `-o DIR` или `--output=DIR`: задает директорию `DIR` для сохранения выходных файлов (CFG, CG). По умолчанию – рядом с входными файлами. Имена имеют следующий формат: “sourceName.functionName.ext”

- `-t` или `--operation tree`: указывает на необходимость вывода в DOT дерево операций вместе с графом потока управления.
- `INPUT_FILES`: список входных файлов для анализа.

Основные функции программы:

- Анализ входных файлов: для каждого входного файла вызывается `parseMyLangFromFile`, результат сохраняется в `MyLangResult`.
- Подготовка «программы»: `buildProgram` создает объект `Program`, объединяя результаты парсинга и строя CFG в два этапа (нахождение определений всех функций и построение CFG для них с нахождением логических ошибок и предупреждений, об этом ниже).
- Генерация CFG: для каждой функции в программе генерируется граф потока управления, который сохраняется в формате DOT.
- Генерация CG: если функция `main` найдена и нет критических ошибок, строится граф вызовов функций.
- Вывод результатов: графы сохраняются в указанных пользователем директориях или по умолчанию рядом с входными файлами.

Детали реализации:

- Для парсинга используется модуль, разработанный в первой работе с помощью ANTLR.
- После построения CFG и сбора списка функций программа выводит ошибки и предупреждения, которые были получены в ходе создания CFG
- Если функция `main` не существует, то будет выведена ошибка и CG не будет создан

Используемые структуры:

```
typedef struct FilesToAnalyze {
    uint32_t filesCount;
    char **fileName;
    MyLangResult **result;
} FilesToAnalyze;
```

Содержит информацию о файлах, которые необходимо проанализировать. Поля:

- `uint32_t filesCount`: общее количество файлов для анализа.
- `MyLangResult **result`: массив указателей на результаты парсинга каждого файла.
- `char **fileName`: массив строк с именами файлов.

```
typedef struct Program {
    FunctionInfo *functions;
    ProgramErrorInfo *errors;
    ProgramWarningInfo *warnings;
} Program;
```

Представляет собой программу после синтаксического и семантического анализа. Поля:

- `FunctionInfo *functions`: связанный список функций в программе.
- `ProgramErrorInfo *errors`: список ошибок, обнаруженных в ходе анализа.
- `ProgramWarningInfo *warnings`: список предупреждений.

```
typedef struct FunctionInfo {
    char *fileName;
    char *functionName;
    TypeInfo *returnType;
    ArgumentInfo *arguments;
    CFG *cfg;
    struct FunctionInfo *next;
    uint32_t line;
    uint32_t pos;
} FunctionInfo;
```

Хранит информацию об отдельной функции внутри программы. Используемые поля (полное описание в другом пункте):

- `char *functionName`: имя функции.
- `char *fileName`: имя файла, где функция определена.

- `CFG *cfg`: граф потока управления функции.
- `FunctionInfo *next`: указатель на следующую функцию в списке.

```
typedef struct CallGraph {
    FunctionNode *functions;
} CallGraph;
```

Представляет граф вызовов функций внутри программы. Поля:

- `FunctionNode *functions`: список узлов функций в графе вызовов.

3.2 Описание модуля для построения и работы с CFG

Используемые структуры:

```
typedef enum {
    CONDITIONAL,
    UNCONDITIONAL,
    TERMINAL
} BlockType;
```

Определяет тип базового блока в графе потока управления (CFG).

- `CONDITIONAL`: базовый блок, содержащий условные переходы.
- `UNCONDITIONAL`: базовый блок с безусловным переходом.
- `TERMINAL`: конечный блок без последующих переходов.

```
typedef enum {
    TRUE_CONDITION,
    FALSE_CONDITION,
    UNCONDITIONAL_JUMP,
} EdgeType;
```

Определяет тип ребра в CFG между базовыми блоками.

- `TRUE_CONDITION`: Ребро, соответствующее выполнению условия (истинное ветвление).
- `FALSE_CONDITION`: Ребро, соответствующее невыполнению условия (ложное ветвление).
- `UNCONDITIONAL_JUMP`: Безусловный переход между блоками.

```
typedef struct {
    char *text;
    OperationTreeNode *otRoot;
} Instruction;
```

Представляет инструкцию внутри базового блока. Поля:

- `char *text`: Текстовое представление инструкции.
- `OperationTreeNode *otRoot`: Корневой узел дерева операций (Operation Tree) для данной инструкции.

```
typedef struct __attribute__((packed)) Edge {
    EdgeType type;
    char *condition;
    struct BasicBlock *fromBlock;
    struct BasicBlock *targetBlock;
    struct Edge *nextOut;
    struct Edge *nextIn;
} Edge;
```

Представляет ребро (переход) между базовыми блоками в CFG. Поля:

- `EdgeType type`: Тип ребра (условное или безусловное).
- `char *condition`: Условие перехода в виде строки (*не используется*).
- `BasicBlock *fromBlock`: Указатель на исходный базовый блок.
- `BasicBlock *targetBlock`: Указатель на целевой базовый блок.
- `Edge *nextOut`: Указатель на следующее исходящее ребро из `fromBlock`.
- `Edge *nextIn`: Указатель на следующее входящее ребро в `targetBlock`.

```
typedef struct BasicBlock {
    int id;
    BlockType type;
    Instruction *instructions;
    int instructionCount;
    int instructionCapacity;
    char *name;
    bool isEmpty;
    bool isBreak;
    Edge *outEdges;
    Edge *inEdges;
    struct BasicBlock *next;
} BasicBlock;
```

Представляет базовый блок («толстый», т. е. может вмещать в себя несколько инструкций, подробности в примерах) в CFG. Поля:

- `int id`: Уникальный идентификатор блока.
- `BlockType type`: Тип базового блока.
- `Instruction *instructions`: Массив инструкций внутри блока.

- `int instructionCount`: Текущее количество инструкций.
- `int instructionCapacity`: Максимальная вместимость массива инструкций.
- `char *name`: Имя блока (может использоваться для отладки или визуализации).
- `bool isEmpty`: Флаг, указывающий, пустой ли блок.
- `bool isBreak`: Флаг, указывающий, является ли блок точкой выхода из цикла (`break`).
- `Edge *outEdges`: Список исходящих ребер.
- `Edge *inEdges`: Список входящих ребер.
- `BasicBlock *next`: Указатель на следующий базовый блок в CFG.

```
typedef struct {
    BasicBlock *entryBlock;
    BasicBlock *blocks;
} CFG;
```

Представляет граф потока управления для функции. Поля:

- `BasicBlock *entryBlock`: Входной (стартовый) базовый блок функции.
- `BasicBlock *blocks`: Список всех базовых блоков в CFG.

```
typedef struct ArgumentInfo {
    TypeInfo *type;
    char *name;
    struct ArgumentInfo *next;
    uint32_t line;
    uint32_t pos;
} ArgumentInfo;
```

Информация об аргументе функции. Поля:

- `TypeInfo *type`: Тип аргумента (структура `TypeInfo` определяет информацию о типе данных).
- `char *name`: Имя аргумента.
- `ArgumentInfo *next`: Указатель на следующий аргумент в списке.
- `uint32_t line`: Номер строки в исходном коде, где объявлен аргумент.
- `uint32_t pos`: Позиция в строке исходного кода.

```
typedef struct FunctionInfo {
    char *fileName;
    char *functionName;
    TypeInfo *returnType;
    ArgumentInfo *arguments;
    CFG *cfg;
    struct FunctionInfo *next;
    uint32_t line;
    uint32_t pos;
} FunctionInfo;
```

Информация о функции в программе. Поля:

- `char *fileName`: Имя файла, в котором определена функция.
- `char *functionName`: Имя функции.
- `TypeInfo *returnType`: Возвращаемый тип функции.
- `ArgumentInfo *arguments`: Список аргументов функции.
- `CFG *cfg`: Граф потока управления функции.
- `FunctionInfo *next`: Указатель на следующую функцию в программе.
- `uint32_t line`: Номер строки в исходном коде, где определена функция.
- `uint32_t pos`: Позиция в строке исходного кода.

```
typedef struct FilesToAnalyze {
    uint32_t filesCount;
    char **fileName;
    MyLangResult **result;
} FilesToAnalyze;
```

Информация о файлах, которые необходимо проанализировать. Поля:

- `uint32_t filesCount`: Количество файлов для анализа.
- `char **fileName`: Массив имен файлов.
- `MyLangResult **result`: Массив результатов парсинга каждого файла.

```
typedef struct __attribute__((packed)) ProgramErrorInfo {
    char *message;
    struct ProgramErrorInfo *next;
} ProgramErrorInfo;
```

Информация об ошибках, возникших при анализе программы. Поля:

- `char *message`: Сообщение об ошибке.

- `ProgramErrorInfo *next`: Указатель на следующую ошибку в списке.

```
typedef struct __attribute__((packed)) ProgramWarningInfo {
    char *message;
    struct ProgramWarningInfo *next;
} ProgramWarningInfo;
```

Информация о предупреждениях, возникших при анализе программы. Поля:

- `char *message`: Сообщение о предупреждении.
- `ProgramWarningInfo *next`: Указатель на следующее предупреждение в списке.

```
typedef struct Program {
    FunctionInfo *functions;
    ProgramErrorInfo *errors;
    ProgramWarningInfo *warnings;
} Program;
```

Представляет полную информацию о программе после анализа. Поля:

- `FunctionInfo *functions`: Список всех функций в программе.
- `ProgramErrorInfo *errors`: Список ошибок программы.
- `ProgramWarningInfo *warnings`: Список предупреждений программы.

Детали реализации:

1) Основная функция для построения базовых блоков

```
BasicBlock *parseBlock(MyAstNode *block, Program *program, const char *filename, bool
isLoop, BasicBlock *prevBlock, BasicBlock *existingBlock, BasicBlock *loopExitBlock,
CFG *cfg, uint32_t *uid);
```

Рекурсивно парсит блок кода (`MyAstNode *block`) и строит соответствующие базовые блоки в CFG, обрабатывая различные конструкции языка, такие как объявления переменных, условные операторы, циклы и выражения.

Может переиспользовать пустые входные блоки без создания новых. Проходится по всем дочерним узлам AST-блока и в зависимости от типа узла (`VAR, BLOCK, IF, WHILE, DO_WHILE, BREAK, EXPR`) вызываются соответствующие функции парсинга. Возвращает указатель на текущий базовый блок после обработки всех узлов. Обработка утверждения и операторов управления:

- VAR: Обработка объявления переменных через parseVar.
- BLOCK: Рекурсивный вызов parseBlock для вложенных блоков.
- IF: Вызов parseIf для обработки условных операторов.
- WHILE и DO_WHILE: Вызов parseWhile и parseDoWhile соответственно для обработки циклов.

Обработка BREAK:

- Если внутри цикла (isLoop == true), добавляется безусловное ребро к loopExitBlock (т. е. к блоку, который следует после цикла).
- Если вне цикла, добавляется сообщение об ошибке в ProgramErrorInfo.
- Проверяется наличие кода после break и, если он есть, генерируется ошибка о недостижимом коде.

Обработка выражений (EXPR):

- Выражения обрабатываются функцией parseExpr, результат добавляется в текущий базовый блок.

2) Функция для обработки всех входящих файлов

```
Program *buildProgram(FilesToAnalyze *files, bool debug);
```

Строит структуру Program, содержащую информацию обо всех функциях, их CFG и обнаруженные ошибки/предупреждения, на основе результатов парсинга файлов.

Осуществляется проход по всем файлам и их функциям. Для каждой функции создается FunctionInfo с информацией о типе возвращаемого значения, аргументах и местоположении. Перед добавлением функции в программу проверяется, не была ли она уже объявлена ранее. В случае повторного объявления добавляется ошибка в ProgramErrorInfo.

Если повторных объявлений не обнаружено, для каждой функции строится CFG – создается стартовый базовый блок START; вызывается `parseBlock` для тела функции, что возвращает последний базовый блок после обработки; добавляется конечный базовый блок END, если последний блок не является терминальным; проверяется наличие возврата из функции и при необходимости добавляются предупреждения.

Построенные `FunctionInfo` с заполненными CFG добавляются в список функций программы и затем указатель на структуру `Program` возвращается.

3) Пример функции для обработки условного оператора (`parseIf`)

Функция `parseIf` предназначена для парсинга условного оператора `if` из абстрактного синтаксического дерева (AST) и построения соответствующих базовых блоков в CFG. Принцип следующий:

1) Создаётся `conditionBlock`, он содержит условие if и имеет два нисходящих ребра:
- Ребро `TRUE_CONDITION` ведет в `thenBlock`.
- Ребро `FALSE_CONDITION` ведет в `elseBlock` или `emptyBlock`, если else отсутствует.
2) Ветви then и else:
- Каждый из блоков ветви может содержать свои базовые блоки, построенные через `parseBlock`.
- После выполнения ветви происходит безусловный переход в `emptyBlock`.
3) Объединяющий блок `emptyBlock`:
- Собирает вместе выходы из обеих ветвей и позволяет продолжить построение CFG после конструкции if.

3.3 Описание модуля для построения Operation Tree

Модуль предоставляет функции и структуры для построения и управления деревом операций (Operation Tree), которое представляет синтаксические и семантические конструкции исходного кода. Для обозначения операций используются константы:

- LIT_READ: Чтение литерала ("litRead")
- READ: Операция чтения ("read")
- WRITE: Операция записи ("write")
- OT_CALL: Вызов функции ("call")
- INDEX: Операция индексации массива ("index")
- DECLARE: Объявление переменной ("declare")
- SEQ_DECLARE: Последовательное объявление ("seqDeclare")
- WITH_TYPE: Объявление с указанием типа ("withType")
- CUSTOM: Пользовательский тип ("custom")
- BUILTIN: Встроенный тип ("builtin")
- OT_ARRAY: Массив ("array")
- RETURN: Операция возврата из функции ("return")
- OT_BREAK: Операция выхода из цикла ("break")

Используемые структуры:

```
typedef struct OperationTreeNode {
  struct OperationTreeNode **children;
  uint32_t childCount;
  const char *label;
  uint32_t line;
  uint32_t pos;
  bool isImaginary;
} OperationTreeNode;
```

Узел дерева операций, представляющий операцию или выражение в коде. Поля:

- `OperationTreeNode **children`: Массив указателей на дочерние узлы
- `uint32_t childCount`: Количество дочерних узлов

- `const char *label`: Метка узла, определяющая тип операции
- `uint32_t line`: Номер строки в исходном коде, соответствующей этому узлу
- `uint32_t pos`: Позиция в строке исходного кода
- `bool isImaginary`: Флаг, указывающий, является ли узел воображаемым (созданным для удобства представления, но не имеющим прямого соответствия в коде)

```
typedef struct TypeInfo {
    char *typeName;
    bool custom;
    bool isArray;
    uint32_t arrayDim;
    uint32_t line;
    uint32_t pos;
    TypeInfo *next;
} TypeInfo;
```

Информация о типе данных переменной или выражения. Поля:

- `char *typeName`: Имя типа
- `bool custom`: Флаг, указывающий, является ли тип пользовательским
- `bool isArray`: Флаг, указывающий, является ли тип массивом
- `uint32_t arrayDim`: Размерность массива (для многомерных массивов)
- `uint32_t line`: Номер строки в исходном коде, где объявлен тип
- `uint32_t pos`: Позиция в строке исходного кода
- `TypeInfo *next`: Указатель на следующий тип в списке (для случаев, когда типы объединены в список)

Также используется структура для ошибок, аналогичная предыдущим.

Детали реализации:

1) Функция для построения ОТ для выражений (expr)

```
OperationTreeNode *buildExprOperationTreeFromAstNode(MyAstNode* root, bool isLvalue,
bool isFunctionName, OperationTreeErrorContainer *container, const char* filename);
```

Рекурсивно строит дерево операций из AST-узла, представляющего выражение. Обрабатывает различные типы выражений, включая присваивания, вызовы функций, индексирование, бинарные и унарные операции, идентификаторы и литералы.

1) **Обработка присваивания (ASSIGN)**:

- Создается узел WRITE с двумя дочерними узлами: левостороннее и правостороннее выражения.
- Проверяется корректность использования присваивания в контексте.

2) **Обработка вызова функции (FUNC_CALL)**:

- Создается узел call с именем функции и списком аргументов.
- Проверяется корректность использования вызова функции в левой части присваивания.

3) **Обработка индексирования (INDEXING)**:

- Создается узел index, представляющий операцию доступа к элементу массива.
- Проверяется наличие индексов и корректность их использования.

4) **Обработка бинарных операций**:

- Проверяется, не используется ли результат бинарной операции в левой части присваивания или в качестве имени функции.
- Создается узел с меткой операции и двумя операндами.

5) **Обработка унарных операций**:

- Аналогично бинарным операциям, но с одним операндом.

6) **Обработка идентификаторов (IDENTIFIER)**:

- Если это левостороннее значение или имя функции, возвращается узел идентификатора.
- Иначе создается узел READ, представляющий чтение значения переменной.

7) **Обработка литералов**:

- Проверяется корректность использования литералов в присваиваниях и вызовах функций.

- Создается узел litRead, содержащий тип и значение литерала.

8) Обработка ошибок:

- При обнаружении некорректных конструкций формируются сообщения об ошибках, добавляемые в OperationTreeErrorContainer.

2) Функция построения ОТ для переменных:

```
OperationTreeNode *buildVarOperationTreeFromAstNode(MyAstNode* root,
OperationTreeErrorContainer *container, TypeInfo* varType, const char* filename);
```

Строит дерево операций для объявления переменных из AST-узла. Определяет количество переменных, объявляемых одновременно. Для одной переменной использует `buildVarDeclareHelper` для создания узла `DECLARE`. Для нескольких переменных создает узел `SEQ_DECLARE`, где каждый дочерний узел — отдельное объявление переменной.

3) Вспомогательная функция для построения ОТ для переменных:

```
OperationTreeNode *buildVarDeclareHelper(MyAstNode* id, MyAstNode* init,
OperationTreeErrorContainer *container, TypeInfo* varType, const char* filename);
```

- Создает узел `DECLARE`, представляющий объявление переменной с возможной инициализацией.
- Включает информацию о типе через вызов `buildTyperefHelper`.
- Обрабатывает как простые, так и массивные типы.
- Если присутствует инициализация, добавляет узел `WRITE` для присваивания начального значения.

3.4 Модуль для работы с Call Graph

Представляет структуры и функции для работы с CG. Построение осуществляется путём обхода всех CFG всех функций и поиска в ОТ каждого блока операции call с помощью функции в модуле для CFG, процесс описываться здесь не будет т. к. он достаточно простой.

Используемые структуры:

```
typedef struct FunctionNode {
    char *functionName;
    struct FunctionNode *next;
```

```
    struct CallEdge *outEdges;
    struct CallEdge *inEdges;
} FunctionNode;
```

Представляет узел в графе вызовов функций, соответствующий конкретной функции в программе. Поля:

- `char *functionName`: Имя функции
- `FunctionNode *next`: Указатель на следующий узел функции в списке всех функций графа
- `CallEdge *outEdges`: Список исходящих ребер (вызовы других функций из данной функции)
- `CallEdge *inEdges`: Список входящих ребер (вызовы данной функции из других функций)

```
typedef struct CallEdge {
    FunctionNode *caller;
    FunctionNode *callee;
    struct CallEdge *nextOut;
    struct CallEdge *nextIn;
} CallEdge;
```

Представляет ребро (вызов функции) между двумя функциями в графе вызовов. Поля:

- `FunctionNode *caller`: Указатель на узел функции, из которой происходит вызов (вызывающая функция)
- `FunctionNode *callee`: Указатель на узел вызываемой функции
- `CallEdge *nextOut`: Указатель на следующее исходящее ребро из вызывающей функции
- `CallEdge *nextIn`: Указатель на следующее входящее ребро в вызываемую функцию

```
typedef struct CallGraph {
    FunctionNode *functions;
} CallGraph;
```

Представляет граф вызовов функций для всей программы. Поля:

- `FunctionNode *functions`: Указатель на связный список всех функций в графе вызовов

4 Примеры

1) Большая программа с блоками, вложенными циклами, условными операторами и break

```
int main(string[] args) {
    int sampleInt;
    sampleInt = 1;
    bool sampleBool = true;
    int[] arr;
    a[1];
    a;
    a();
    true;
    1;
    1+1;
    "str";

    {
        int k = 0;
        int c;
        {
            sampleInt = 1;
            bool sampleBool = true;
            {
                int k = 0;
                int c;
                int c;
                int c;
                o;
            }
        }
    }

    {
        int k = 0;
    }

    if (a + 4) {
        int y;
    } else {
        int u;
    }

    if (false) {
        int y;
    } else {
        int u;
    }


    do {
        int y;
        while (a + b) {
            int b;
            int y;
```

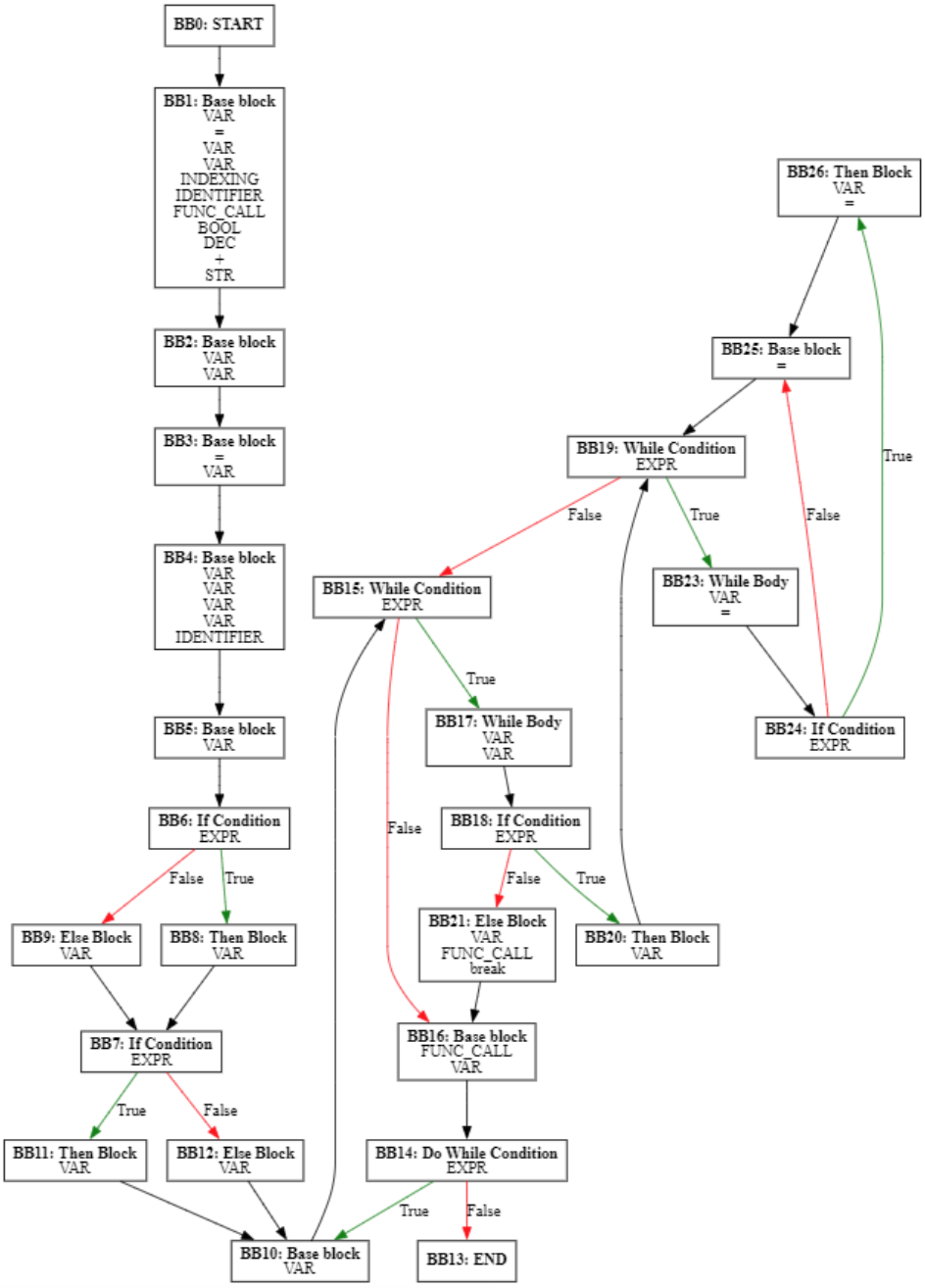
```
            if (false) {
                int y;
            } else {
                int u;
                call();
                break;
                call();
            }
            while (a + b) {
                int f;
                a = 8 + 9;
                if (false) {
                    int y;
                    a = 9;
                }
                a = 9;
            }
        }
        check();
        int k;
    } while (true);

}

int test(string[] args) {
    int l;
}
```

Вывод программы:

```
Errors:
Control error. Unreachable code after break at ./inputs/cfg/exampleInput:58:17

Warnings:
No return warning. Can't use instruction at ./inputs/cfg/exampleInput:78:9 as a
return value
No return warning. Can't use instruction at ./inputs/cfg/exampleInput:73:14 as a
return value
```

Выводится ошибка о том, что существует недостижимый код после break, а также выводятся предупреждения, что в функциях нет возвращаемого значения.

Полученный граф управления для main (отображение деревьев операций отключено для компактности):



Базовые блоки «толстые» - создание новых происходит только при создании в коде блока {}, при использовании условного оператора, при использовании циклов. Т. е. пока есть возможность все идущие подряд инструкции будут записаны в текущий блок. Для наглядности внутри блока отображаются условные названия инструкций из AST.

Как можно видеть – при использовании условного оператора поток управления после then-else переходит сразу в следующий непустой базовый блок. Последний блок в циклах передаёт управление в условный блок цикла.

При использовании break управление переходит в тот же блок, что и при false переходе из уловного блока цикла, что корректно. При этом break работает корректно во вложенных циклах, что демонстрирует данный пример.

При использовании do_while управление сначала переходит в тело цикла, а затем уже после тела в условие цикла, что корректно. Между собой условные операторы и циклы взаимодействуют корректно.

2) Пример CG для двух файлов, где функции вызывают друг друга

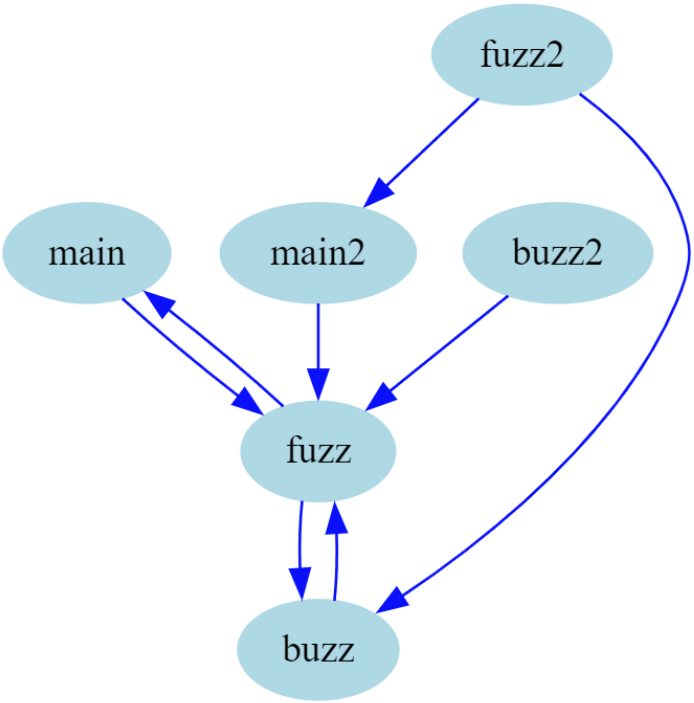
Код первого файла:

```
int main(string[] args) {
    fuzz(true, 0, "testString");
    0;
}

bool buzz() {
    fuzz();
    a[1];
}

fuzz(bool b, int k, myType mt) {
    (buzz(k)(a))(s, k);
    int k = main() + 1;
    0;
}
```

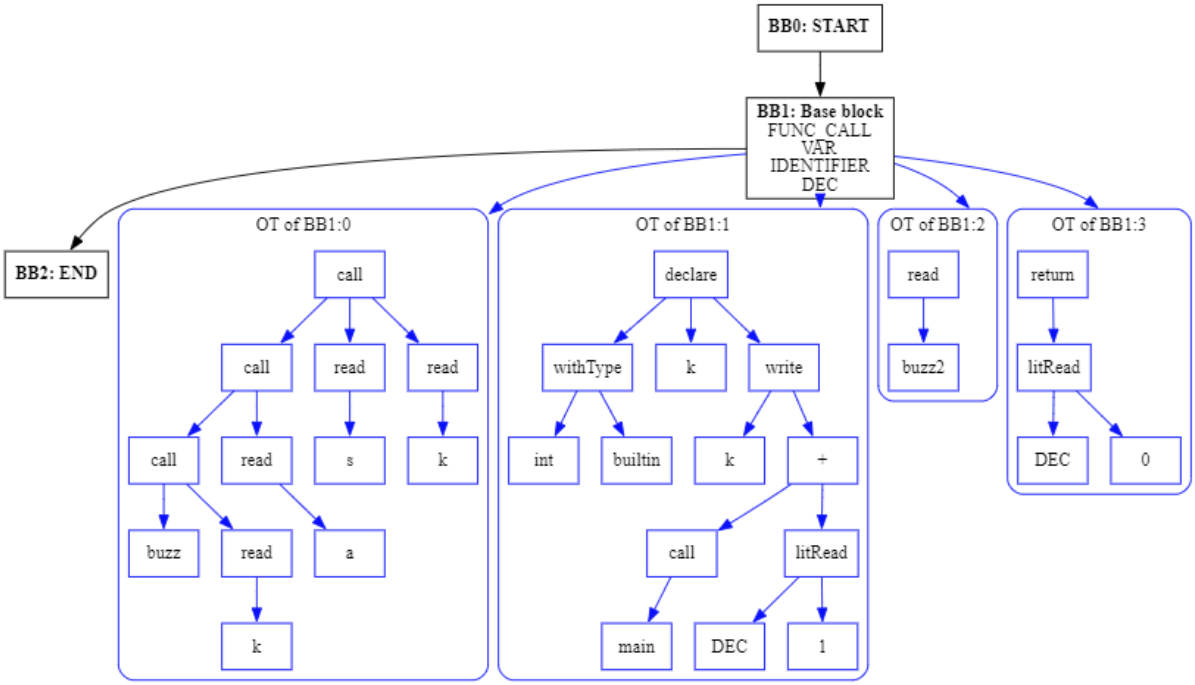
Код второго файла (см. имена функций):

```
int main2(string[] args) {
    fuzz2(true, 0, "testString");
    0;
}

bool buzz2() {
    fuzz();
    a[1];
}

fuzz2(bool b, int k, myType mt) {
    (buzz(k)(a))(s, k);
    int k = main() + 1;
    buzz2;
    0;
}
```

Полученный граф вызовов (Call Graph):



Граф потока управления функции fuzz2 с деревьями операций (для примера):



Здесь можно видеть вложенный вызов функций (даже если нет имени), объявление и инициализацию переменной (не уверен, что это было нужно, но

в будущем можно таким образом составить таблицу символов при обходе), и возврат значения (т. к. литерал может быть возвращаемым значением).

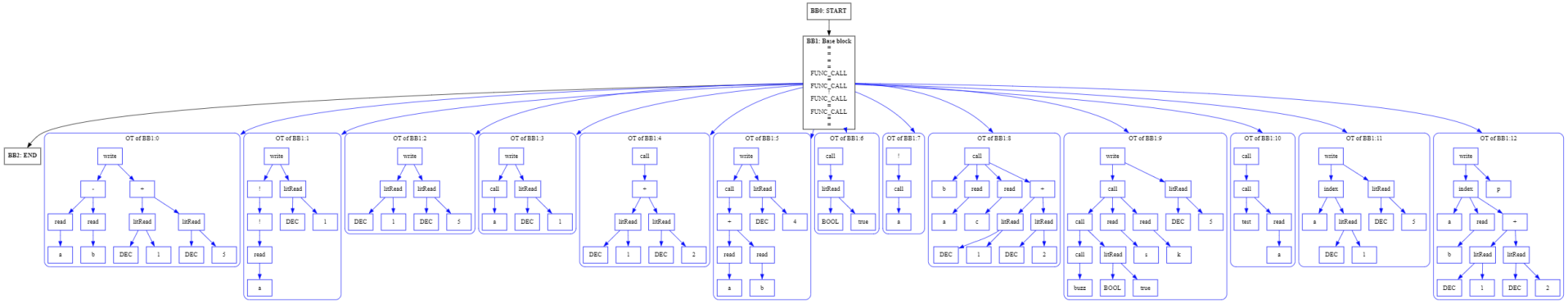
3) Пример 1 для деревьев операций

```
int main(string[] args) {
    a - b = 1 + 5;
    !!a = 1;
    1 = 5;
    a() = 1;
    (1+2)();
    (a+b)() = 4;
    true();
    !a();
    b(a, c, 1+2);
    (buzz()(true))(s, k) = 5;
    test(a)();
    a[1] = 5;
    a[b, (1+2)] = p[];
}
```

Смешанный пример с логическими ошибками и некоторыми интересными конструкциями. Вывод программы:

```
Errors:
Index error. Missing index value at ./inputs/cfg/otTest:14:19

Assign error. Can't use function calling to assign at ./inputs/cfg/otTest:11:6

Call error. Can't use literal to call function at ./inputs/cfg/otTest:8:5

Assign error. Can't use function calling to assign at ./inputs/cfg/otTest:7:7

Call error. Can't use binary operation to call function at ./inputs/cfg/otTest:7:7

Call error. Can't use binary operation to call function at ./inputs/cfg/otTest:6:7

Assign error. Can't use function calling to assign at ./inputs/cfg/otTest:5:5

Assign error. Can't use literal to assign at ./inputs/cfg/otTest:4:5

Assign error. Can't use unary operation result to assign at ./inputs/cfg/otTest:3:5

Assign error. Can't use binary operation result to assign at ./inputs/cfg/otTest:2:7

Warnings:
No return warning. Can't use instruction at ./inputs/cfg/otTest:14:17 as a return
value
```

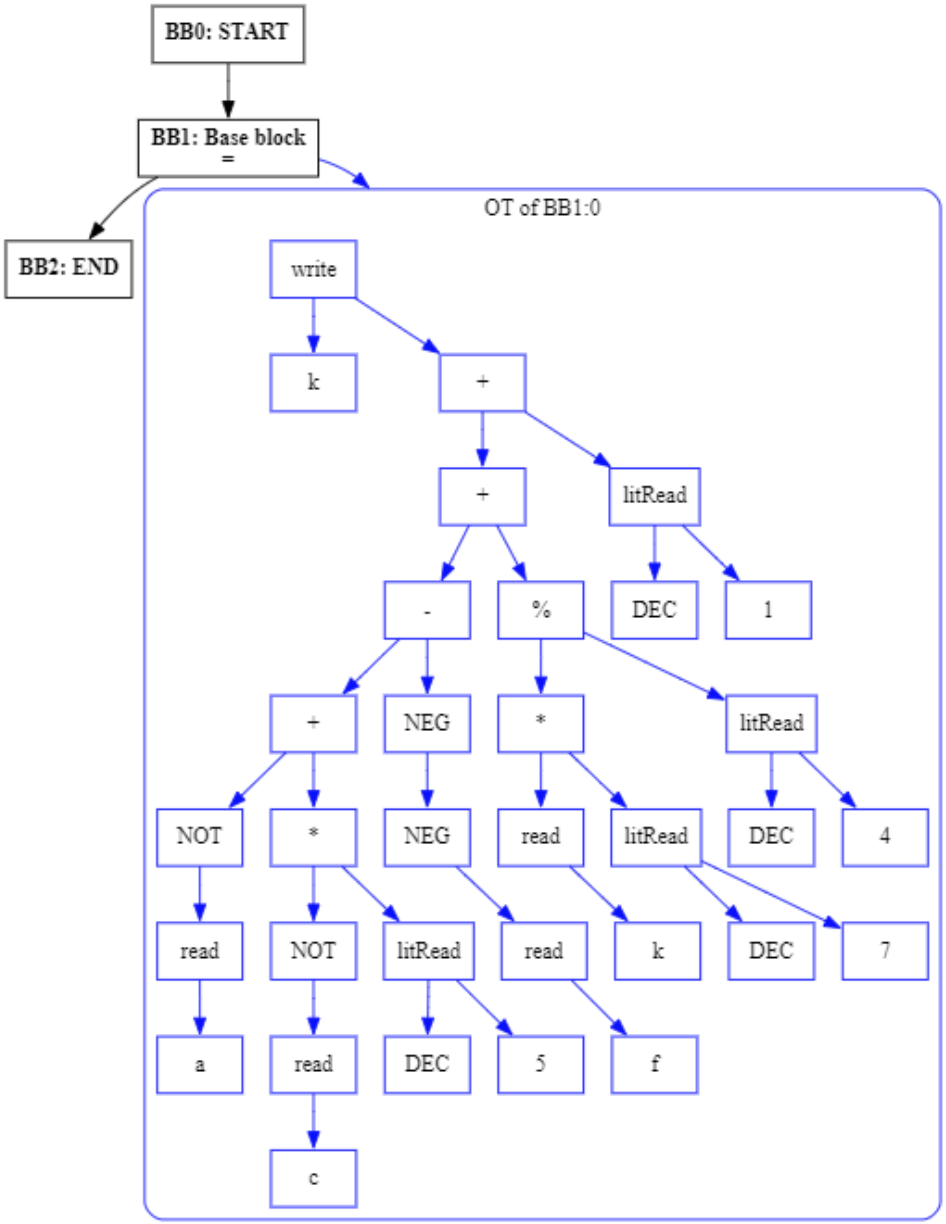
Ошибки разного смысла, описание и места в коде приведены, комментировать не буду. Деревья операций и граф всё равно строиться, но, естественно, деревья операций не все будут корректны с точки зрения смысла:



4) Пример 2 для деревьев операций

```
int main(string[] args) {
    k = !a + !c * 5 - --f + k * 7 % 4 + 1;
}
```

Код представляет собой математическое выражение. Операции обрабатываются, как и в AST – с учётом приоритета. Если необходимо прочитать литерал, то для этого используется соответствующая инструкция с указанием типа литерала. Полученный граф и дерево операций:


BB0: START
BB1: Base block
=
BB2: END
OT of BB1:0
write
k
+
+
litRead
-
%
DEC
1
+
NEG
*
litRead
NOT
*
NEG
read
litRead
DEC
4
read
NOT
litRead
read
k
DEC
7
a
read
DEC
5
f
c

5 Выводы

В ходе выполнения задания были разработаны модули для построения и анализа графа потока управления (CFG), дерева операций (Operation Tree) и графа вызовов функций (Call Graph) для «моего» языка программирования. Основные цели задания заключались в создании инструментов для базового семантического анализа программ, а также в визуализации внутренних структур кода для облегчения отладки и дальнейшего развития компилятора.

Для достижения поставленных целей были решены следующие задачи:

1) **Разработка структур данных**

   Были спроектированы и реализованы структуры данных для представления базовых блоков, ребер в CFG, узлов дерева операций и узлов графа вызовов функций. Это обеспечило основу для эффективного хранения и обработки информации о программе.

2) **Реализация основных функций**

   Были разработаны функции для построения CFG из AST, включая обработчики различных конструкций языка (`if`, `while`, `do-while`, `break` и т. д.). Также были реализованы функции для построения дерева операций из AST, позволяющие детально анализировать выражения и операции в коде.

3) **Обработка ошибок и предупреждений**

   В процессе построения CFG и дерева операций внедрена система сбора и обработки ошибок и предупреждений, что позволяет получать подробную информацию о возможных проблемах в коде.

4) **Визуализация внутренних структур**

   Реализованы функции для экспорта CFG и графа вызовов функций в формат DOT, что позволяет визуализировать эти структуры с помощью сторонних инструментов, таких как Graphviz.