Национальный исследовательский университет ИТМО
Мегафакультет компьютерных технологий и управления
Факультет программной инженерии и компьютерной техники

Отчет по практическому заданию №2

по курсу «Языки программирования»

Вариант: 1

Руководитель: Ю. Д. Кореньков

Группа: P4119

Выполнил студент: Мохаммад Кусае Альосман

13 Января 2026 г.

Санкт-Петербург
2026

Оглавление

Задание 2

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. 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 труктуры 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

1. Цель работы

Реализовать построение графа потока управления (CFG) на основе результатов синтаксического анализа набора входных файлов, выполнить анализ полученной информации и сформировать набор файлов с графическим представлением графов потока управления для подпрограмм, а также графа вызовов между ними.

2. План работы

1. Описать структуры данных для представления набора входных файлов, подпрограмм и графов потока управления (включая базовые блоки и деревья операций).
2. На основе деревьев разбора, получаемых из модуля, разработанного в задании 1, реализовать модуль построения графа потока управления и регистрации логических ошибок.
3. Разработать тестовую программу, которая:
    a. читает из аргументов командной строки имена входных файлов и выходную директорию,
    b. вызывает модуль из задания 1 для получения деревьев разбора,
    c. использует модуль построения графов потока управления для всех подпрограмм,
    d. сохраняет графы потока управления и граф вызовов в выходные файлы.

4. Реализовать тестовую программу для демонстрации работоспособности созданного модуля.

3. Ход работы

1. Подготовил окружение: установил Node.js (npm/npx), Python 3.13, а также библиотеку Graphviz (утилита `dot`) и Python-пакеты `tree_sitter` и `graphviz`.
2. Сгенерировал парсер по грамматике Tree-sitter для варианта 1 командой `npx tree-sitter generate` в каталоге `tree-sitter`.
3. Собрал динамическую библиотеку грамматики (DLL) из `parser.c` с помощью компилятора GCC, получив файл `var1.dll`.
4. Запустил анализатор на тестовых программах варианта 1 через `python/main.py`, указав путь к DLL и входные файлы; результаты сохранились в каталог `out`.
5. Проверил корректность разбора и типизации:
   5.1. отсутствие файла `out/call_graph.errors.txt` означает отсутствие синтаксических ошибок разбора,
   5.2. отсутствие файла `out/type_errors.txt` означает отсутствие ошибок типизации.
6. Открыл сформированные графы управления/вызовов в формате SVG из `out/graph/` и убедился, что ветвления и циклы отображаются корректно.

6.1. Результаты работы (файлы):

6.2. дерево разбора: out/tree/<имя_файла>

6.3. граф управления/вызовов: out/graph/<имя_файла>_main.svg

6.4. ошибки разбора (если есть): out/call_graph.errors.txt

6.5. ошибки типизации (если есть): out/type_errors.txt

6.6.

6.7. Описание разработанных структур данных

В рамках выполнения данного задания будут использоваться структуры данных, приведенные ниже.

В отличии от лабораторной работы 1, где дерево было использовано напрямую из библиотеки tree-sitter, в этой я добавил класс-обертку, который позволит упростить работу с деревом:

```
@dataclass
class TreeViewNode:
    """
    Узел логического дерева, соответствующий тому, что раньше выводилось в файл.

    label  — то, что раньше писалось в строку (identifier, "(", ...).
    node   — исходный node из парсера (для "настоящих" узлов).
    children — потомки в отображаемом дереве.
    """

    label: str
    node: Any | None
    children: List["TreeViewNode"]
```

Таким образом у меня имеется доступ к тексту элемента, его наследникам и, если необходимо, к его непосредственно tree-sitter корню.

После формирования для очередного файла данной структуры, необходимо сформировать граф потока управления функций, которые есть в этом файле. Каждый граф формируется отдельно.

Структура, используемая для формирования графа приведена ниже:

```
@dataclass
class Block:
    """
    Один блок графа (будущий прямоугольник на картинке).

    id      - уникальный номер блока
    label   - текст внутри блока (или можно класть сюда указатель на дерево операций)
    succs   - список исходящих рёбер вида (id_следующего_блока, метка_ребра)
              метка_ребра, например, "True"/"False" для if.
    """
    id: int
    label: str
    tree: TreeViewNode = field(default=None)
    succs: List[Tuple[int, Optional[str]]] = field(default_factory=list)


@dataclass
class CFG:
    """
    Весь граф потока управления.
    """
    blocks: Dict[int, Block] = field(default_factory=dict)
    next_id: int = 0  # счётчик для выдачи свежих id
    errors: List[str] = field(default_factory=list)
    call_names: set[str] = field(default_factory=set)
```

Класс Blok служит одной структурной единицей графа, он имеет свой уникальный id (для построения графа), свой label и дерево операций, которое

состоит из ранее рассмотренного TreeViewNode, но немного переформатированного., а также из исходящих ребер.

В случае, если указано дерево, оно будет отрисовано, в противном случае отображаться будет label, такие блоки используются для удобства формирования графа, на постобработке от них можно было бы избавиться т.к. они не несут в себе функционала, однако, было принято решение оставить их т.к. они не мешают.

Класс CFG является хранилищем всего графа, он предоставляет интерфейс для генерации блоков, а также хранить промежуточную информацию об ошибках, возникающих в ходе формирования графа, а также об вызываемых в функции других функциях.

Результатом всех этапов работы, выполненных для каждого файла, для каждой функции в них служит словарь, в котором каждому имени функции сопоставляются вызовы в ней, ошибки, непосредственно граф потока управления и дерево этой функции, они будут использованы в дальнейшем, в 3 лабораторной работе для определения типов данных.

6.8. Интерфейс разработанной программы

Для работы с программой был обновлен интерфейс, представленный в лабораторной работе 1, была добавлена возможность указать несколько входных файлов, а выходной теперь указывается не файл, а директория, куда помещаются все файлы, полученные в ходе проекта.

Для подробностей о запуске программы был добавлен флаг -h:

```
usage: lab_2_main.py [-h] [--lib LIB_PATH] [--lang LIB_LANG_NAME] rest [rest ...]

Сборка и запуск tree-sitter парсера

positional arguments:
  rest                  Режим 1 (--lib): file1 [file2 ...] out_dir
                        Режим 2: grammar_dir lang_name file1 [file2 ...] out_dir

options:
  -h, --help            show this help message and exit
  --lib LIB_PATH        Путь к уже скомпилированной tree-sitter библиотеке
                        (.so/.dll/.dylib). В этом режиме grammar_dir и lang_name
                        позиционно не задаются.

  --lang LIB_LANG_NAME  Имя языка при использовании --lib (например, 'foo' для
                        tree_sitter_foo). Если не указано, будет выведено из имени
                        файла библиотеки.
```

Имеется возможность указать путь к уже скомпилированной библиотеке и задать её название (без названия она определяется по имени файла библиотеки), если же библиотека не задана ожидается путь и её название для компиляции, после чего указываются файлы и в конце директория для результатов работы программы.

Вот пример консольной команды для запуска программы с уже скомпилированным файлом библиотеки:

```
python .\python\lab_2_main.py --lib .\build\var2.dll
.\examples\all_structures_and_expr .\examples\empty .\examples\functions  out
```

Здесь, указан путь до библиотеки, затем файлы для сборки и непосредственно директория для результатов.

Результатом работы программы является следующее дерево файлов:

```
out
├── tree
│   ├── files
│   └── ...
├── graph
│   ├── files.svg
│   └── ...
├── call_graph.svg
└── call_graph.errors.txt
```

В папке tree хранятся все созданные деревья, кроме содержащих ошибки, все ошибки в ходе процесса создания дерева будут сохранены в файл call_graph.errors.txt.

В папке graph хранятся созданные svg картинки с графами потока управления, в случае возникновения какой-то ошибки при построении графа, он не сохранится, однако ошибка об этом будет отображена в call_graph.errors.txt. Если же ошибка возникнет в ходе разбора деревьев операций, граф будет отображаться, но соответствующая ошибка будет выведена в узел, а также добавлена в call_graph.errors.txt.

На данный момент невозможность отрисовать граф потока управления возникает только в случае возникновения break вне цикла, в остальном же такие ситуации успешно обрабатываются в первом этапе формирования дерева.

В дереве же операций ошибки могут возникать, если использовать операции вызова функции, присваивания или обращения по индексам к чему-то кроме переменных.

В call_graph.svg будет нарисована картинка графа переходов, а также отображены все не созданные функции и функции с ошибками.

6.9. Пример работы программы

Для варианта 1 подготовлен набор тестовых программ в каталоге examples_v1. Каждая программа демонстрирует отдельную конструкцию языка (объявление переменных, if/else, while, do-while, break, вызовы функций, вложенные конструкции и выражения).

Важно: перегрузка функций в реализации не поддерживается. Поэтому при запуске сразу нескольких файлов в одном вызове имена функций не должны повторяться (в частности, в нескольких файлах не должно быть функции main). Для удобства тестирования каждый пример запускается отдельно либо функции переименовываются (например, main_01, main_02 и т.д.).

Команда запуска (Windows PowerShell) для одного входного файла:

C:\Users\...\Python313\python.exe .\python\main.py --lib var1 .\tree-sitter\var1.dll .\examples_v1\01_basic .\out

Где:

• --lib var1 — имя языка (должно совпадать с именем, передаваемым в tree-sitter Language).

• .\tree-sitter\var1.dll — скомпилированная библиотека tree-sitter для грамматики варианта 1.

• .\examples_v1\01_basic — входной файл с программой.

• .\out — выходная директория для результатов.

После запуска формируются следующие результаты:

• out\tree\<имя_файла> — дерево разбора (TreeView). Если в дереве есть подстроки ERROR или MISSING, во входном тексте есть синтаксические проблемы.

• out\graph\<имя_файла>_<функция>.svg — граф потока управления (CFG) для каждой функции.

• out\call_graph.svg — граф вызовов функций.

• out\call_graph.errors.txt — файл создаётся только при наличии ошибок построения/логических ошибок (например, break вне цикла или повторное определение функции).

• out\type_errors.txt — файл создаётся только при наличии ошибок строгой типизации.

• out\result.asm — ассемблерный вывод (если предыдущие этапы прошли без критических ошибок).

Проверка корректности результата выполняется по двум пунктам:

1) Отсутствуют файлы out\call_graph.errors.txt и out\type_errors.txt (или они пустые) — это означает, что CFG построен без зарегистрированных ошибок и строгая типизация прошла успешно.

2) Визуальная проверка CFG (SVG):

• Для if/else должен быть узел-условие с двумя исходящими рёбрами (True/False) и последующим слиянием ветвей.

• Для while должен присутствовать переход назад к проверке условия (петля).

• Для do-while тело выполняется минимум один раз: сначала блок тела, затем проверка условия с переходом назад.

• Для break внутри цикла — переход к завершающему блоку цикла (end_while / end_do).

• В узлах операций корректно отображаются store/load/const/call и бинарные операции.

Ниже приведены примеры полученных графов (файлы формируются в out\graph, их удобно открывать в браузере или в VS Code через встроенный просмотрщик SVG):

• v1_test_main.svg — пример с var + if/else + while + do-while;

• 03_calls_main.svg — пример вызовов функций;

• 06_nested_main.svg — пример вложенных циклов и условий.

```
method hello_world()
begin
    true;
    false;
    "asd";
    'a';
    0x123;
    0b1010;
    123;
    b + 3;
    asd;
    !g;
    hello_world();
    a[a,b,c];
    (3);

    if a then a;
    if a then a; else a;
    begin end;
    while a do begin end;
    repeat begin end; while a;
    repeat begin end; until 1;
    repeat begin break; end; while a;


    if a then
        if b then
            a := a + 2;
        else
            b := 2;

end;
```

```
method hello_world_2();
method hello_world_3(a, b: int, c: a): array[,,] of a;
method c()
var a,b,c:int; d,e; begin end;
method hello_world_2()
var a,b,c:int; d,e; begin end;
```

```
method hello_world_2();
method hello_world_3(a, b: int, c: a): array[,,] of a;
method c()
var a,b,c:int; d,e; begin end;
method hello_world_2()
var a,b,c:int; d,e; begin
    c();
    g();
end;
```

4. Вывод

В ходе выполнения лабораторной работы был разработан и реализован модуль построения графа потока управления (CFG) на основе синтаксических деревьев, получаемых из парсера, созданного в первой лабораторной работе. Были спроектированы и реализованы структуры данных для представления логического дерева (TreeViewNode), базовых блоков (Block) и всего графа потока управления (CFG), а также предусмотрены механизмы накопления информации об ошибках и вызовах подпрограмм. На основе этих структур формируются CFG для каждой функции, а также сопутствующие данные, которые будут использованы в последующих этапах (в частности, в третьей лабораторной работе для анализа типов данных).

Была создана тестовая программа (python/main.py), позволяющая обрабатывать один или несколько входных файлов, подключать сгенерированную библиотеку tree-sitter (var1.dll), строить для каждой функции граф потока управления (CFG) и общий граф вызовов функций. Результаты сохраняются в выходную директорию: текстовые деревья разбора в out/tree, CFG-графы в формате SVG в out/graph, общий граф вызовов out/call_graph.svg; при наличии ошибок дополнительно формируются out/call_graph.errors.txt и out/type_errors.txt, а при успешной типизации сохраняется ассемблерный код out/result.asm.

Проведённые эксперименты на тестовых примерах показали работоспособность разработанного модуля, корректность построения графов потока управления и графа вызовов, а также адекватную регистрацию и вывод ошибок. Таким образом, цель лабораторной работы достигнута: создан модуль построения CFG и графа вызовов, интегрированный с синтаксическим анализатором и обеспечивающий основу для дальнейшего статического анализа программ.