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: 2

Выполнил студент группы P4119 _____ Д.Л. Симоновский
(подпись)

Руководитель _____ Ю. Д. Кореньков
(подпись)

18 ноября 2025 г.

Санкт-Петербург
2025

Оглавление

1. Цель работы 2

2. План работы 2

3. Ход работы 2

3.1. Описание разработанных структур данных 2

3.2. Интерфейс разработанной программы 4

3.3. Пример работы программы 6

4. Вывод 13

5. Исходные файлы 14

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

3.1. 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.

3.2. 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.

3.3. Пример работы программы

Подадим в программу следующий ввод, который охватывает все конструкции языка:

Листинг 3.1. Исходный текст программы

```
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;
```

Ниже приведены отрывки итогового графа потока управления поскольку весь рисунок занимает много места, его можно найти в репозитории.

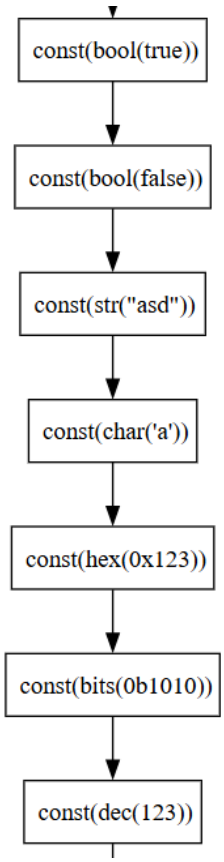

Рис. 3.1. Граф потока управления для констант.

Выше приведены все возможные константы для данной синтаксической модели. Изначально предполагалось перевести все числа в десятичную систему, однако позже было принято отказаться от этого и произвести данное преобразование на более позднем этапе.

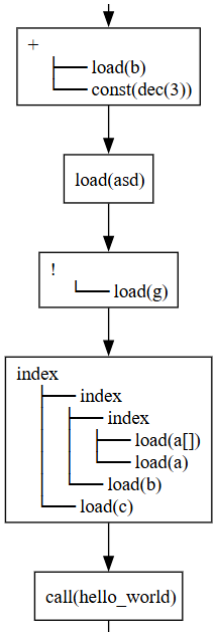

Рис. 3.2. Граф потока управления для expression.

Выше представлен граф потока управления для всех expression, как можно заметить, index представлен интересным способом, сначала загружается адрес массива a, после чего дополнительно загружаются все необходимые индексы, что позволит сформировать интересующий адрес.

Все остальные операции очевидны.

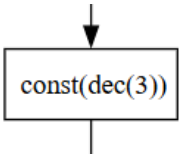

Рис. 3.3. Граф потока управления для скобочного выражения.

Последнее не показанное выражение – скобочное, однако оно вырождается т.к. лишь задает приоритизацию.

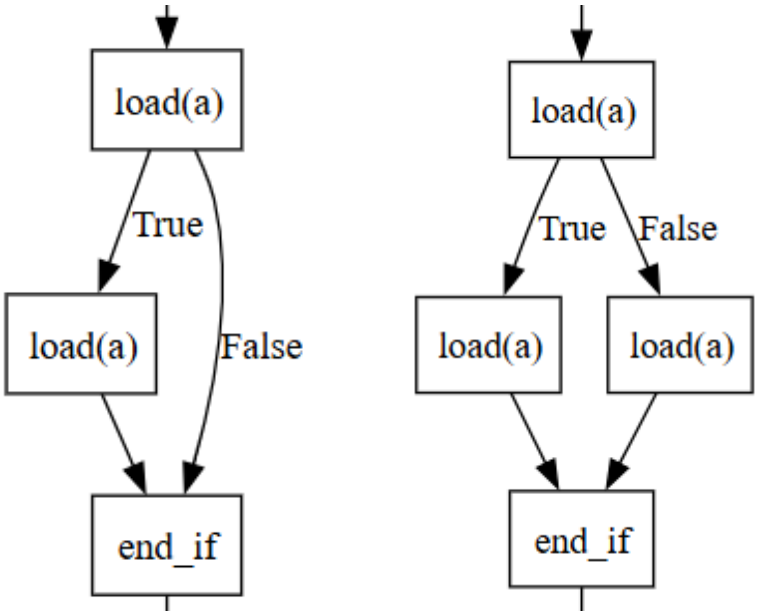

Рис. 3.4. Граф потока управления для условных выражений.

Как видно на рисунке выше, условные выражения также корректно обрабатываются, как в случае наличия else ветки, так и без неё.

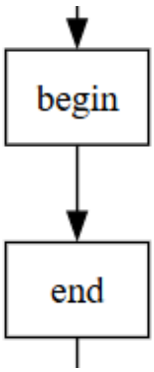

Рис. 3.5. Граф потока управления для begin-end конструкций.

Данные конструкции не являются обязательными и лишь помогают формировать граф, однако они не были удалены т.к. упрощают чтение графа.

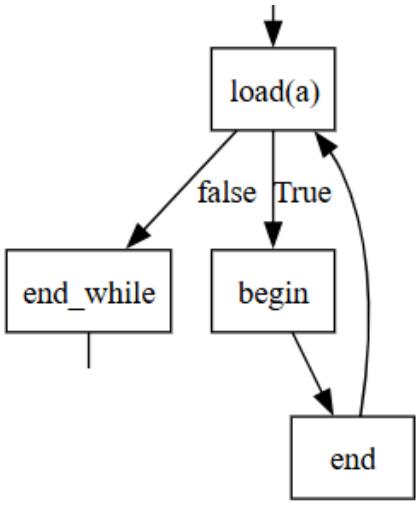

Рис. 3.6. Граф потока управления для while конструкции.

На рисунке выше блок от end_while был убран, чтоб не сбивать с толку, но как можно заметить, конструкция построена верно. В случае, если выражение true, операции продолжают выполнение, а когда false, прекращают его.

Стоит отметить, что end_while также, как и begin end является вспомогательным блоком, но удален он не был.

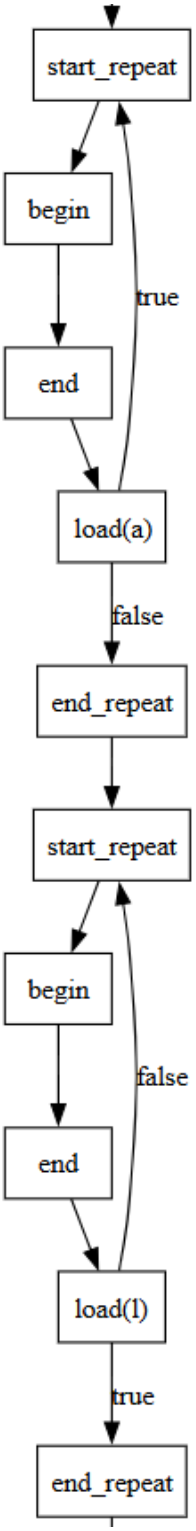

Рис. 3.7. Граф потока управления для do конструкции.

Выше представлено две do конструкции, одна с do while, вторая же с do until, как можно заметить, они одинаковые, а отличаются лишь условиями перехода.

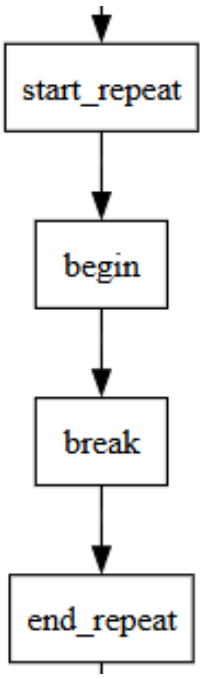

Рис. 3.8. Граф потока управления для break конструкции.

Последняя рассматриваемая конструкция – break. Как можно заметить, она сразу перешла к концу цикла, также стоит отметить, что так как условие из-за брейка никогда не проверяется в потоке управления бы образовались блоки, к которым никто не ведет, а они идут в end_repeat, однако я их почистил, дополнительным проходом.

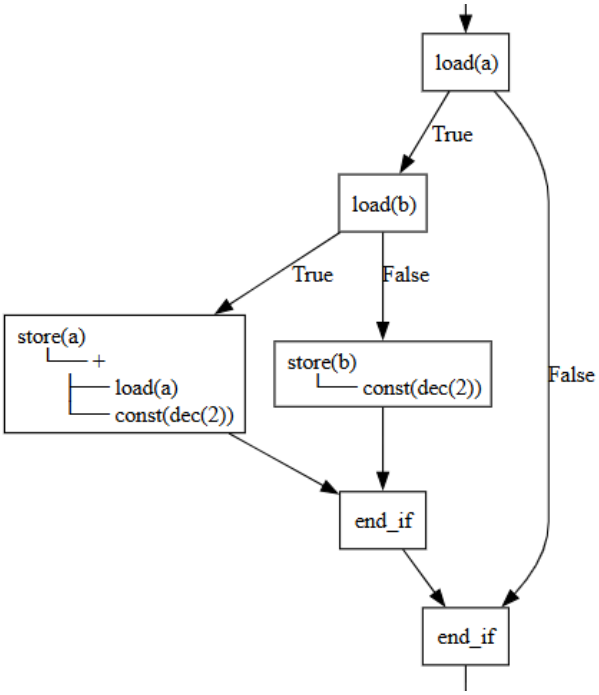

Рис. 3.9. Граф потока управления для сложной конструкции.

В конце приведен граф потока управления для сложной конструкции с висящим else, как можно заметить он корректно обрабатывается.

Далее рассмотрим граф вызовов:

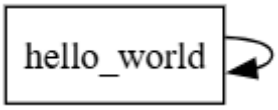

Рис. 3.10. Граф вызовов.

Как можно заметить, получилась одна функция, которая вызывает сама себя. Добавим еще несколько функций, для этого подключим файл с различными определениями функций:

```
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;
```

Интересно отметить, что тут есть объявления функций, без тела, а также функция, которая реализует объявленную ранее. В итоге получим такой граф переходов:

Рис. 3.11. Граф вызовов многих функций.

Как мы видим, они никак не соединены, добавим в hello_world_2 вызов функции c и не определенной у нас функции g:

```
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;
```

Выполним запуск и получим обновленный граф переходов:

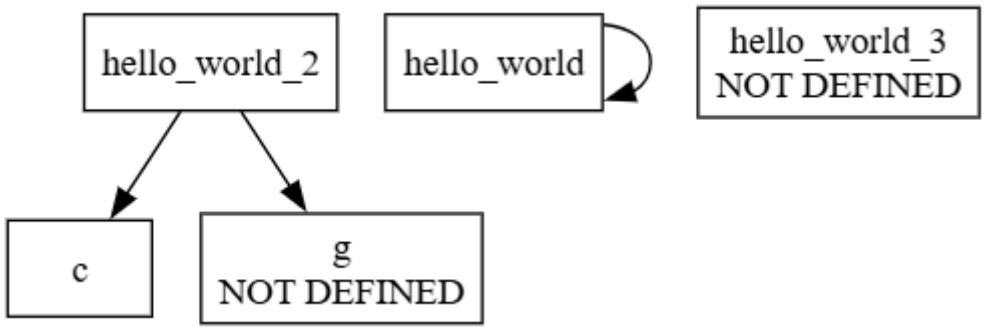

Рис. 3.12. Обновленный граф вызовов многих функций.

Как можно заметить, добавились ребра между hello_world_2 и c, а также появился блок g, который не определен.

Также важно отметить, что перегрузка в данной реализации программы не предусмотрена и вызывает ошибку, еще хочется указать на то, что объявление функции обязано быть раньше её реализации, иначе также будет ошибка.

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, позволяющая обрабатывать несколько входных файлов, задавать исходную библиотеку tree-sitter или компилировать её при запуске, а также сохранять результаты работы в виде дерева каталогов с синтаксическими деревьями, графами потока управления и графом вызовов. Программа корректно обрабатывает широкий спектр конструкций языка: константы, выражения (включая индексные и скобочные), условные операторы с

и без else, блоки begin–end, циклы while и repeat (в вариантах while/until), а также оператор break с дополнительной очисткой недостижимых блоков. Реализована генерация графа вызовов с фиксацией неописанных функций, функций без тела и ошибок, связанных с нарушением порядка объявления и отсутствием поддержки перегрузки.

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.

5. Исходные файлы

Исходные файлы проекта: репозиторий [Электронный ресурс]. — GitHub. — Режим доступа: https://github.com/DafterT/parsing_grammar (дата обращения: 20.10.2025).