Федеральное государственное автономное образовательное учреждение высшего образования

Университет ИТМО

Дисциплина: Системное программное обеспечение

Лабораторная работа 2

Выполнили:
Кривоносов Егор Дмитриевич

Группа: P4114

Преподаватель:
Кореньков Юрий Дмитриевич

2024 г.

Санкт-Петербург

Оглавление

Задание

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 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

Используемые структуры данных

```
struct GraphConfig {
    char *procedureName;
    Block *entryblock;
    BlockList *finalblocks;
    int nextId;
};

struct GraphConfigBuilder {
    BlockList *after_loop_block_stack;
    BlockList *curr_loop_guard_stack;
    Block *current_block;
    int current_id;
    GraphConfig *cfg;
};

struct Link {
    Block *source;
    Block *target;
    char *comment;
};

struct LinkList {
    Link **links;
    int count;
};

struct BlockList {
    Block **blocks;
    int count;
};
```

При обходе нод в дереве я формирую конфигурацию графа - GraphConfig. Обход дерева осуществляется с помощью DFS. После конфигурирования графа потока управления создаются файлы формата .txt с данными для построения графического отображения. С помощью утилиты Graphviz (dot). Генерируется SVG-фал с графическим отображением для функции.

Примеры построения графов управления

Пример 1

Функция:

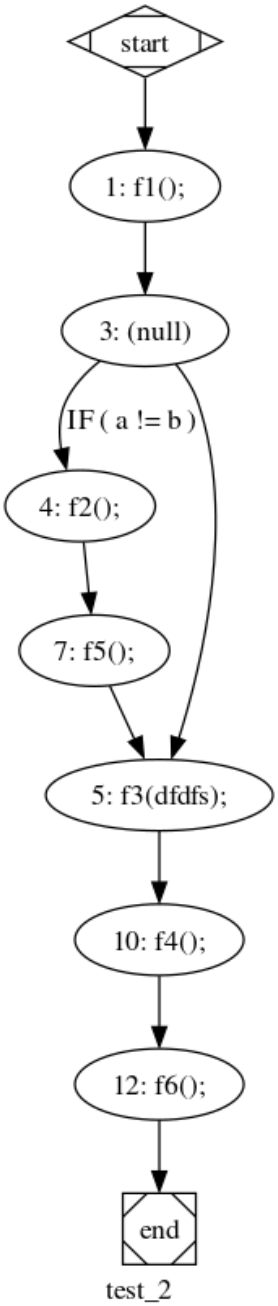
```
bool test_2() {
    f1();
    if (a != b) {
        f2();
        f5();
    }
    f3("dfdfs");
    f4();
    f6();
}
```

Полученный файл для генерации:

```
digraph G {label=test_2;
"1: f1();" -> "3: (null)"[label=""];
"3: (null)" -> "4: f2();"[label="IF ( a != b )"];
"4: f2();" -> "7: f5();"[label=""];
"7: f5();" -> "5: f3(dfdfs);"[label=""];
"5: f3(dfdfs);" -> "10: f4();"[label=""];
"10: f4();" -> "12: f6();"[label=""];
"12: f6();" -> end;
"3: (null)" -> "5: f3(dfdfs);"[label=""];

start -> "1: f1();";
start [shape=Mdiamond]; end [shape=Msquare];
}
```

Дерево:



Пример 2

Функция:

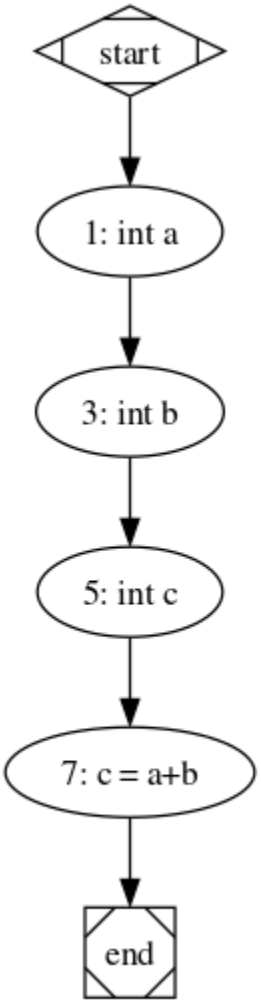
```
bool test_3() {
    int a;
    int b;
    int c;
    c = a + b;
}
```

Полученный файл для генерации:

```
digraph G {label=test_3;
"1: int a" -> "3: int b"[label=""];
"3: int b" -> "5: int c"[label=""];
"5: int c" -> "7: c = a+b"[label=""];
"7: c = a+b" -> end;

start -> "1: int a";
start [shape=Mdiamond]; end [shape=Msquare];
}
```

Дерево:



test_3

Пример 3

Функция:

```
bool test_4(int arg_1 ) {
    f1();
    do {
        f2();
        if (a > b){
            f3();
            // f31();
        } else {
            f4();
            while (x < y) {
                f5();
                f6();
            };
            f7();
            while (i == j) {
                while (x == y) {
                    while (m == n) {
                        f8();
                    };
                };
            };
            f11();
        };
        f12();
        f13();
        f14(12);
        f15();
    } while (j > 3);
    f16();
}
```

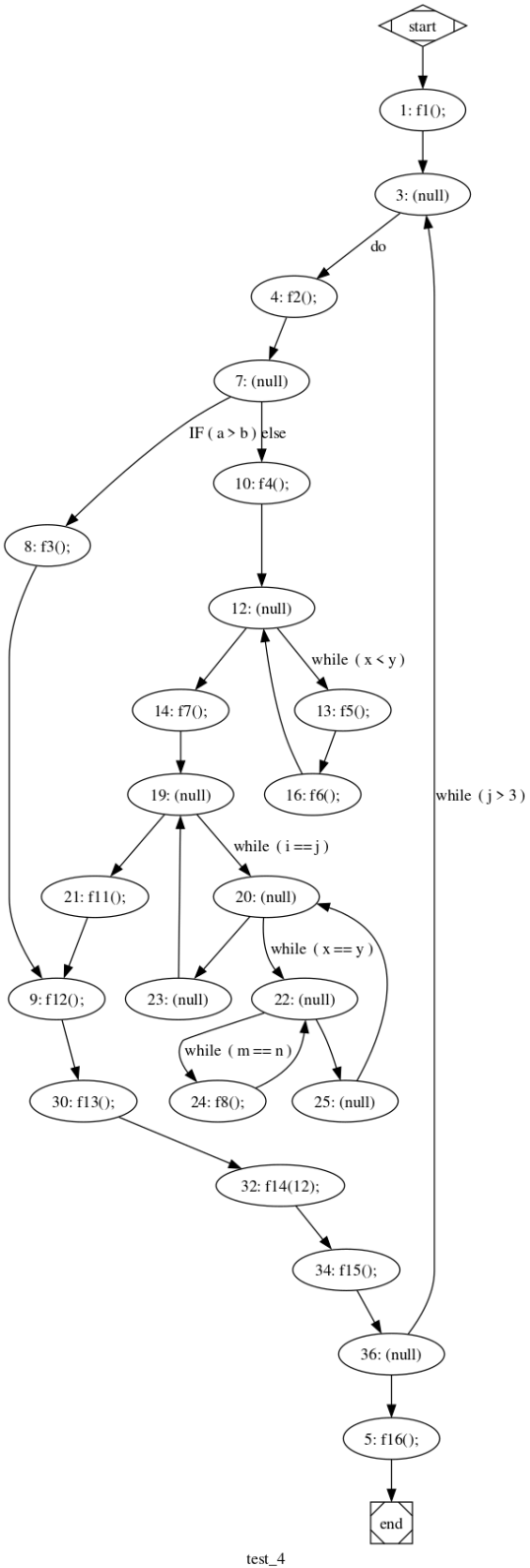
Полученный файл для генерации:

```
digraph G {label=test_4;
"1: f1();" -> "3: (null)"[label=""];
"3: (null)" -> "4: f2();"[label="do"];
"4: f2();" -> "7: (null)"[label=""];
"7: (null)" -> "8: f3();"[label="IF ( a > b )"];
"8: f3();" -> "9: f12();"[label=""];
"9: f12();" -> "30: f13();"[label=""];
"30: f13();" -> "32: f14(12);"[label=""];
```

```
"32: f14(12);" -> "34: f15();"[label=""];
"34: f15();" -> "36: (null)"[label=""];
"36: (null)" -> "3: (null)"[label="while ( j > 3 )"];
"36: (null)" -> "5: f16();"[label=""];
"5: f16();" -> end;
"7: (null)" -> "10: f4();
"[label="else"];
"10: f4();
" -> "12: (null)"[label=""];
"12: (null)" -> "13: f5();"[label="while ( x < y )"];
"13: f5();" -> "16: f6();"[label=""];
"16: f6();" -> "12: (null)"[label=""];
"12: (null)" -> "14: f7();"[label=""];
"14: f7();" -> "19: (null)"[label=""];
"19: (null)" -> "20: (null)"[label="while ( i == j )"];
"20: (null)" -> "22: (null)"[label="while ( x == y )"];
"22: (null)" -> "24: f8();"[label="while ( m == n )"];
"24: f8();" -> "22: (null)"[label=""];
"22: (null)" -> "25: (null)"[label=""];
"25: (null)" -> "20: (null)"[label=""];
"20: (null)" -> "23: (null)"[label=""];
"23: (null)" -> "19: (null)"[label=""];
"19: (null)" -> "21: f11();"[label=""];
"21: f11();" -> "9: f12();"[label=""];

start -> "1: f1();";
start [shape=Mdiamond]; end [shape=Msquare];
}
```

Дерево:



test_4

Пример 4

Функция:

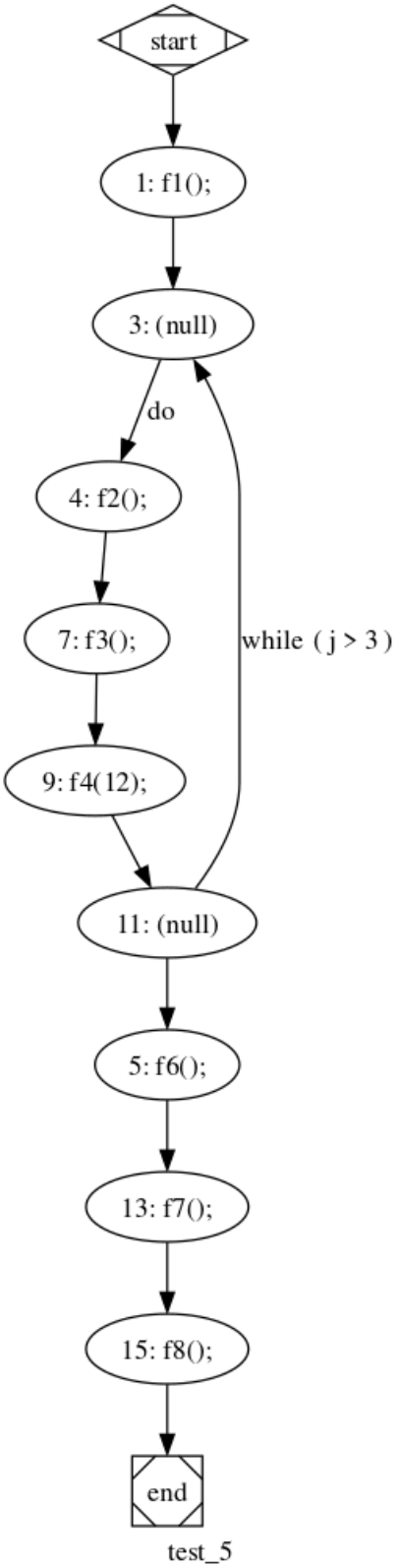
```
bool test_5(int arg_1 ) {
    f1();
    do {
        f2();
        f3();
        f4(12);
    } while (j > 3);
    f6();
    f7();
    f8();
}
```

Полученный файл для генерации:

```
digraph G {label=test_5;
"1: f1();" -> "3: (null)"[label=""];
"3: (null)" -> "4: f2();"[label="do"];
"4: f2();" -> "7: f3();"[label=""];
"7: f3();" -> "9: f4(12);"[label=""];
"9: f4(12);" -> "11: (null)"[label=""];
"11: (null)" -> "3: (null)"[label="while ( j > 3 )"];
"11: (null)" -> "5: f6();"[label=""];
"5: f6();" -> "13: f7();"[label=""];
"13: f7();" -> "15: f8();"[label=""];
"15: f8();" -> end;

start -> "1: f1();";
start [shape=Mdiamond]; end [shape=Msquare];
}
```

Дерево:


start
1: f1();
3: (null)
do
4: f2();
7: f3();
while (j > 3)
9: f4(12);
11: (null)
5: f6();
13: f7();
15: f8();
end
test_5

Вывод

Во время выполнения лабораторной работы, я познакомился с работой графа потока управления. Также использовал алгоритмы построения на основе AST и поработал с выводом в формате dot для отображения графического представления графа.