ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ АВТОНОМНОЕ
ОБРАЗОВАТЕЛЬНОЕ УЧРЕЖДЕНИЕ ВЫСШЕГО ОБРАЗОВАНИЯ

«Национальный исследовательский университет ИТМО»

Факультет «Программной Инженерии и Компьютерных Технологий»

Отчет о лабораторной работе №2

«Задание 2»

по дисциплине

«Языки Программирования»

Вариант 3

Выполнил:

Студент группы P4119

Эр Мерт

Проверил:

Доцент факультета ПИиКТ, к.т.н.

Кореньков Ю. Д.

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.

Описание разработанных структур данных

BasicBlock: Ключевая единица CFG. Содержит уникальный `id`, список операций (`Operation`) и указатели на следующие блоки: `next` (безусловный переход), `trueSucc` и `falseSucc` (условные переходы).

```
public final class BasicBlock {

    private final int id;
    private final List<Operation> operations = new ArrayList<>();

    public BasicBlock next;

    public BasicBlock trueSucc;

    public BasicBlock falseSucc;

    public BasicBlock(int id) {
        this.id = id;
    }

    public int id() {
        return id;
    }

    public List<Operation> operations() {
        return operations;
    }

    public BasicBlock next() {
        return next;
    }

    public BasicBlock trueSucc() {
        return trueSucc;
    }

    public BasicBlock falseSucc() {
        return falseSucc;
    }
}
```

Operation: Иерархия узлов промежуточного представления (OpAssign, OpBinary, OpCall, OpArraySet и др.).

CfgFunction: Обертка над графом функции, хранящая входной (`entry`) и выходной (`exit`) блоки.

Описание работы

Программа принимает на вход объект `FunctionNode` (результат Лабы №1) и возвращает `CfgFunction`.

Программный интерфейс и особенности реализации

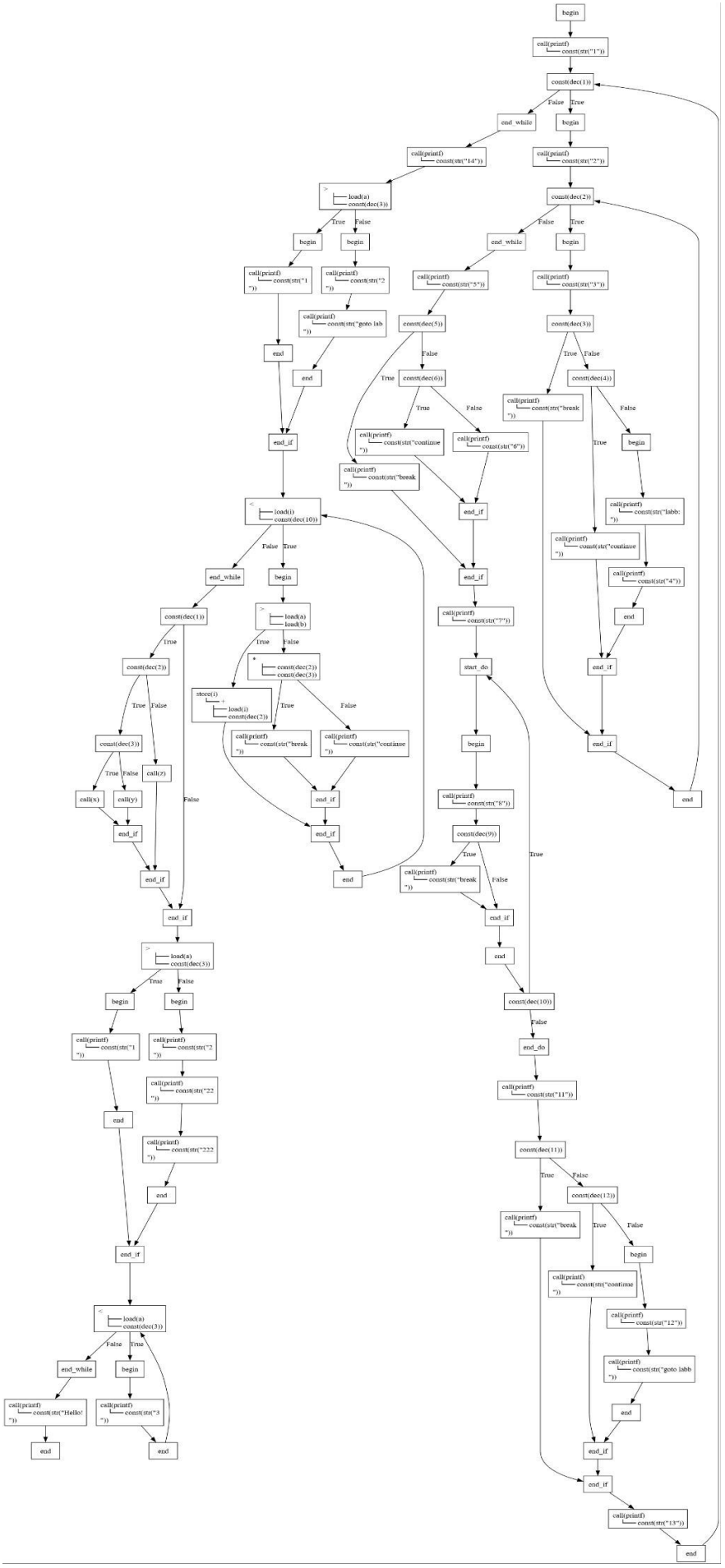
- **Интерфейс:** Основная точка входа — метод `build(FunctionNode fn)`.
- **Алгоритм:** Используется линейный обход стейтментов. При обнаружении управляющей конструкции (например, `IF`) текущий блок "завершается", создаются новые блоки для веток `then` и `else`, а также `joinBlock` для их слияния.
- **Обработка циклов:** Для циклов `WHILE` создается `condBlock`, куда ведут обратные ребра из конца тела цикла.
- **Контекст построения:** Класс `BuildContext` инкапсулирует состояние построения (текущий блок, стек циклов, список всех блоков), что позволяет избежать передачи десятка параметров в каждый метод.
- **Специальная логика:** Присваивание значения переменной, имя которой совпадает с именем функции, интерпретируется как установка возвращаемого значения и ведет к автоматическому переходу в `exit`-блок.

Аспекты реализации

Пример реализации обработки цикла `WHILE`:

```
private void handleWhile(WhileStatementNode whileSt, BuildContext ctx) {

    BasicBlock condBlock = newBlock(ctx.blocks);
    BasicBlock bodyBlock = newBlock(ctx.blocks);
    BasicBlock afterLoop = newBlock(ctx.blocks);

    if (!ctx.terminated.contains(ctx.current)) {
        ctx.current.next = condBlock;
        ctx.terminated.add(ctx.current);
    }

    ctx.loopStack.push(new LoopInfo(condBlock, afterLoop));

    // condition
    ctx.current = condBlock;
    Operation cond = buildExpr(whileSt.condition(), ctx);
    if (cond != null) {
        ctx.current.operations().add(cond);
    }
    ctx.current.trueSucc = bodyBlock;
    ctx.current.falseSucc = afterLoop;
    ctx.terminated.add(ctx.current);

    // body
    ctx.current = bodyBlock;
    for (StatementNode s : whileSt.body()) {
        processStatement(s, ctx);
    }
    if (!ctx.terminated.contains(ctx.current)) {
        ctx.current.next = condBlock;
        ctx.terminated.add(ctx.current);
    }

    ctx.loopStack.pop();
    ctx.current = afterLoop;
}
```

CFG


begin
call(printf)
const(str("1"))
const(dec(1))
False
True
end_while
begin
call(printf)
const(str("14"))
call(printf)
const(str("2"))
>
load(a)
const(dec(3))
const(dec(2))
True
False
False
True
begin
begin
end_while
begin
call(printf)
const(str("1
"))
call(printf)
const(str("2
"))
call(printf)
const(str("5"))
call(printf)
const(str("3"))
call(printf)
const(str("goto lab
"))
const(dec(5))
const(dec(3))
end
end
False
True
False
True
const(dec(6))
const(dec(4))
True
False
True
False
call(printf)
const(str("break
"))
call(printf)
const(str("continue
"))
call(printf)
const(str("6"))
begin
end_if
call(printf)
const(str("break
"))
<
load(i)
const(dec(10))
call(printf)
const(str("labb:
"))
end_if
call(printf)
const(str("continue
"))
False
True
end_while
begin
end_if
call(printf)
const(str("4"))
const(dec(1))
>
load(a)
load(b)
call(printf)
const(str("7"))
end
True
True
False
start_do
const(dec(2))
*
const(dec(2))
const(dec(3))
end_if
store(i)
+
load(i)
const(dec(2))
True
False
begin
const(dec(3))
call(printf)
const(str("brenk
"))
call(printf)
const(str("continue
"))
end_if
True
False
call(x)
call(y)
call(z)
call(printf)
const(str("8"))
end_if
end_if
const(dec(9))
True
False
end
call(printf)
const(str("break
"))
end_if
end
const(dec(10))
False
end_do
call(printf)
const(str("11"))
const(dec(11))
True
False
const(dec(12))
call(printf)
const(str("break
"))
True
False
begin
call(printf)
const(str("continue
"))
call(printf)
const(str("12"))
call(printf)
const(str("goto labb
"))
end
end_if
end_if
call(printf)
const(str("13"))
end

Вывод:

В ходе работы была освоена концепция базовых блоков и управления потоком в компиляторах. Реализованный подход со стеком loopStack позволил элегантно решить задачу нелокальных переходов (break). Построенный CFG является фундаментом для дальнейшей генерации кода, так как он явно определяет все возможные пути исполнения программы.