ГУАП

КАФЕДРА № 44

ОТЧЕТ
ЗАЩИЩЕН С ОЦЕНКОЙ

ПРЕПОДАВАТЕЛЬ

| канд. техн. наук, доцент | | Н.В. Кучин |
|---|---|---|
| должность, уч. степень, звание | подпись, дата | инициалы, фамилия |

ОТЧЕТ О ЛАБОРАТОРНОЙ РАБОТЕ №1

# ГЕНЕРАЦИЯ И ОПТИМИЗАЦИЯ ПРОГРАММНОГО КОДА

по курсу: Системное программное обеспечение

РАБОТУ ВЫПОЛНИЛ

| СТУДЕНТ гр. № 4143 | | Е. Д. Тегай |
|---|---|---|
| | подпись, дата | инициалы, фамилия |

Санкт-Петербург 2024

**Цель работы**

Изучение основных принципов генерации компилятором объектного кода, выполнение генерации объектного кода программы на основе результатов синтаксического анализа для заданного входного языка.

Изучение основных принципов оптимизации компилятором объектного кода для линейного участка программы, ознакомление с методами оптимизации результирующего объектного кода с помощью метода исключения лишних операций.

**Задание по лабораторной работе**

Задание по лабораторной работе варианта №19 имеет следующую формулировку: входной язык содержит арифметические выражения, разделенные символом ; (точка с запятой). Арифметические выражения состоят из идентификаторов, символьных констант (один символ в одинарных кавычках), знака присваивания (:=), знаков операций +, -, *, / и круглых скобок.

Также вариант №19 дополняется содержанием из рисунка 1, где в качестве первого столбца указан номер варианта, второго – номер варианта грамматики и третьего – допустимые лексемы входного языка.

| | | |
|---|---|---|
| 19 | 1 | Идентификаторы, символьные константы (в одинарных кавычках) |

Рисунок 1 – Задание индивидуального варианта

Вариант грамматики, на который ссылается предпоследний столбик из рисунка 1, продемонстрирован на рисунке 2. Следует отметить, что символ S является начальным символом грамматики; S, F, T и E обозначают нетерминальные символы. Терминальные символы выделены жирным шрифтом. Вместо терминального символа **a** должны подставляться лексемы в соответствии с вариантом задания.

*1.* $S \rightarrow \mathbf{a} := F;$
$F \rightarrow F+T \mid T$
$T \rightarrow T*E \mid T/E \mid E$
$E \rightarrow (F) \mid -(F) \mid \mathbf{a}$

Рисунок 2 – Вариант грамматики

**Запись заданной грамматики входного языка в форме Бэкуса-Наура**

Исходная запись продемонстрирована ниже:

$$G(\{S, F, T, E\}, \{a, :=, ;, +, *, /, (, ), -\}, P, S)$$

**Фрагменты объектного кода в виде триад для операций заданной грамматики**

Разберём пример, продемонстрированный ниже.

$$C := A + B * D;$$

Это выражение можно представить в виде триад:

1) (*, B, D) – результат сохраняется как ^0
2) (+, A, ^0) – результат сохраняется как ^1
3) (:=, C, ^1) – присваивание результата второй триады переменной C

Следует пояснить, что триады записаны в формате (операция, первый аргумент, второй аргумент).

**Листинг программы**

Созданная программа является многофайловым проектом. Основной файл ***SPO3.mjs*** представляет собой программу, которая выполняет чтение, обработку и преобразование данных из JSON-файла с синтаксическим деревом, полученным в прошлой лабораторной работе, в ассемблерный код, используя также промежуточную оптимизацию.

В нём сначала импортируются необходимые модули для корректной работы всего проекта и для работы с файловой системой. Программа считывает содержимое файла, где хранится дерево. В случае ошибки при чтении выводится соответствующее сообщение в консоль. В ином случае содержимое файла преобразуется из строки JSON в объект JavaScript. Затем

идёт проверка формата данных – является ли прочитанное дерево массивом (такой формат необходим для корректной работы программы). Для каждой части дерева вызывается соответствующая функция, которая генерирует триады. Искомые триады, как и каждый из результатов функциональной обработки, выводится в виде таблицы с соответствующим заголовком в консоли. Затем идёт оптимизация триад, а также генерация ассемблерного кода.

Всего в проекте существует 3 вспомогательных файла. Разберём суть каждого из них. Первым вспомогательным файлом является файл ***triadGeneration.mjs***. Данный файл, как ясно из названия, реализует функцию для генерации триад из дерева. Внутри него прописаны две функции. Первая из них предназначена для поиска последнего оператора в строке (она будет использоваться второй функцией). Вторая же функция является как бы основной в рамках данного файла. Она принимает узел дерева разбора, создаёт массив, в который будут записываться сгенерированные триады. Внутри себя также содержит рекурсивную функцию, которая выполняет обход дерева и обработку узлов. Для каждого узла, исходя из его типа и содержимого, создаются триады с указанием операции, аргументов и результата. В результате выполнения этой функции из дерева разбора создаётся упорядоченный список триад, который описывают порядок выполнения операций в искомом выражении, сохраняя последовательность вычислений и зависимости между элементами.

Вторым вспомогательным файлом является файл ***optimization.mjs***. является самым большим по своему содержанию файлом. Как ясно из названия, данный файл представляет собой функцию, которая оптимизирует таблицу триад, устраняя дублирующиеся операции, обновляя ссылки и корректируя аргументы для уменьшения количества операций в результирующей структуре. Структура его такова, что сначала инициализируются все необходимые переменные, а также флаги для проверочных операций. Далее идёт основной цикл, который проходится по

всем сгенерированным триадам, затем обновляются ссылки (в случае удаление триад) и возвращается массив, содержащий оптимизированные триады с обновлёнными ссылками и удалёнными дубликатами.

Последним вспомогательным файлом является файл ***assembler.mjs***. как ясно из названия, данный файл представляет собой функцию, которая преобразует триады (оптимизированные) в ассемблерные команды. Структура его такова: сначала идёт инициализация массива для хранения ассемблерных команд и иных необходимых переменных. Затем идёт снова прохождение по уже оптимизированным триадам, из которых извлекаются аргументы с целью дальнейшей проверки, являются ли они корректными (в случае некорректности она пропускается). Далее описана функция, которая проверяет, является ли аргумент ссылкой на другую триаду. Если это так, она возвращает соответствующую временную переменную или сам аргумент. Таким образом, для каждой триады создаётся ассемблерная команда. Функция возвращает массив, который содержит сгенерированные ассемблерные команды.

***Основной файл SPO3.mjs***

```
// Импортируем модуль `fs` для работы с файловой системой, чтобы читать данные из
файла.
import fs from 'fs';
// Импортируем функцию `Triads_Generator` из модуля `triadGeneration.mjs`,
которая отвечает за генерацию триад.
import { Triads_Generator } from './triadGeneration.mjs';
// Импортируем функцию `Optimizator` из модуля `optimization.mjs`, которая
выполняет оптимизацию сгенерированных триад.
import { Optimizator } from './optimization.mjs';
// Импортируем функцию `Assembler_Generator` из модуля `assembler.mjs`, которая
переводит триады в ассемблерный код.
import { Assembler_Generator } from './assembler.mjs';
// Читаем содержимое файла `OutputTree.json` с кодировкой `utf8`, используя
асинхронный метод `readFile`.
// В callback-функцию передаются параметры `error` и `data` — ошибка и содержимое
файла.
fs.readFile('OutputTree.json', 'utf8', (error, data) => {
    // Проверяем, возникла ли ошибка при чтении файла.
    if (error) {
        // Логируем ошибку, если файл не удалось прочитать.
        console.erroror('Error: Failed to read the file', error);
        // Завершаем выполнение функции при ошибке.
```

```
        return;
    }
    // Парсим содержимое файла `data` из формата JSON в объект JavaScript и
сохраняем в переменную `originalTree`.
    const originalTree = JSON.parse(data);
    // Инициализируем пустой массив `findedTriads`, в который будут записаны
сгенерированные триады.
    const findedTriads = [];
    // Создаем объект `index` с ключом `value`, инициализированным значением 1,
для отслеживания индексов триад.
    const index = { value: 1 };
    // Проверяем, является ли `originalTree` массивом, чтобы обработать каждое
дерево отдельно.
    if (Array.isArray(originalTree)) {
        // Перебираем каждое дерево в массиве `originalTree` с помощью `forEach`.
        originalTree.forEach(tree => {
            // Генерируем триады из текущего дерева, используя
`Triads_Generator`, и передаем `index` для отслеживания порядкового номера триад.
            const triads = Triads_Generator(tree, index);
            // Добавляем сгенерированные триады в массив `findedTriads`.
            findedTriads.push(...triads);
        });
        // Выводим разделитель для улучшения читаемости вывода в консоли.
        console.log('------------------------НАЙДЕННЫЕ ТРИАДЫ-------------------
------');
        // Отображаем найденные триады в табличном формате, извлекая свойства
`operation`, `first_arg` и `second_arg`.
        console.table(findedTriads.map(({ operation, first_arg, second_arg }) =>
({
            'ОПЕРАЦИЯ': operation,
            'ПЕРВЫЙ АРГУМЕНТ': first_arg,
            'ВТОРОЙ АРГУМЕНТ': second_arg
        })));
        // Выводим заголовок перед оптимизацией триад.
        console.log('------------------------ОПТИМИЗАЦИЯ-----------------------
-');
        // Оптимизируем триады, используя функцию `Optimizator`, и сохраняем
результат в `finalOptimizedTriads`.
        const finalOptimizedTriads = Optimizator(findedTriads);
        // Отображаем оптимизированные триады в табличном формате.
        console.table(finalOptimizedTriads.map(({ operation, first_arg,
second_arg }) => ({
            'ОПЕРАЦИЯ': operation,
            'ПЕРВЫЙ АРГУМЕНТ': first_arg,
            'ВТОРОЙ АРГУМЕНТ': second_arg
        })));
        // Выводим заголовок перед генерацией ассемблерного кода.
        console.log('------------------------АССЕМБЛЕРНЫЙ КОД-------------------
------');
        // Генерируем ассемблерный код из оптимизированных триад с помощью
`Assembler_Generator`.
```

```
        const assembly = Assembler_Generator(finalOptimizedTriads);
        // Отображаем ассемблерный код в табличном формате, показывая поле
`command`.
        console.table(assembly.map(({ command }) => ({ 'КОМАНДА': command
})));
    } else {
        // Выводим сообщение об ошибке, если формат данных `originalTree` не
является массивом.
        console.error('Error: Invalid tree format');
    }
});
```

### *Вспомогательный файл triadGeneration.mjs*

```
// Функция для поиска последнего оператора в выражении.
function Find_Last_Operator(expression) {
    // Ищем все операторы `+`, `-`, `*`, `/` в строке с помощью регулярного
выражения.
    const matched = expression.match(/[\+\-\*\/]/g);
    // Возвращаем последний найденный оператор или `null`, если операторов нет.
    return matched ? matched[matched.length - 1] : null;
  }

  // Экспортируемая функция для генерации триад, принимающая узел дерева и индекс
(по умолчанию со значением 0).
  export function Triads_Generator(node, index = { value: 0 }) {
    // Инициализируем массив для хранения сгенерированных триад.
    const triads = [];
    // Переменная для хранения последнего результата триады.
    let lastTriadOutput;

    // Вложенная функция для рекурсивной обработки узлов дерева.
    function Tree_Processing(curNode) {
        // Если текущий узел пустой, возвращаем `null`.
        if (!curNode) return null;
        // Если узел является терминалом, возвращаем его лексему.
        if (curNode.type === 'Terminal') {
            return curNode.lexem;
        }
        // Если узел является нетерминалом, обрабатываем его дочерние узлы.
        if (curNode.type === 'NonTerminal') {
            // Обрабатываем каждый дочерний узел и фильтруем скобки.
            const childNodeRes = curNode.ch.map(child =>
Tree_Processing(child)).filter(output => output !== '(' && output !== ')');
            // Переменная для хранения результата текущего узла.
            let nodeRes;
            // Удаляем скобки из лексемы текущего узла.
            const formattedLexem = curNode.lexem.replace(/[()]/g, '');
```

```
            // Проверяем, содержит ли лексема присваивание `:=`.
            if (formattedLexem.includes(':=')) {
                // Генерируем результат узла и увеличиваем значение индекса.
                nodeRes = `^${index.value++ - 1}`;
                // Добавляем триаду с операцией присваивания.
                triads.push({
                    operation: ':=',
                    first_arg: childNodeRes[0],
                    second_arg: lastTriadOutput || childNodeRes[2],
                    output: nodeRes,
                });
                // Обновляем `lastTriadOutput` для использования в последующих
операциях.
                lastTriadOutput = nodeRes;
                // Возвращаем результат текущего узла.
                return nodeRes;
            }
            // Если количество дочерних результатов равно 3, продолжаем
обработку.
            if (childNodeRes.length === 3) {
                // Ищем последний оператор в лексеме.
                const operation = Find_Last_Operator(formattedLexem);
                // Проверяем и корректируем первый элемент, если он — скобка.
                if (childNodeRes[0] === '(') {
                    childNodeRes[0] = childNodeRes[1];
                }
                // Определяем аргументы для триады.
                const first_arg = childNodeRes[0];
                const second_arg = childNodeRes[2];
                // Проверяем, является ли первый аргумент ссылкой на триаду.
                const isFirstArgReference = typeof first_arg === 'string' &&
first_arg.startsWith('^');
                // Форматируем первый аргумент, если он ссылка.
                const firstArg = isFirstArgReference ?
`^${parseInt(first_arg.slice(1))}` : first_arg;
                // Проверяем, что аргументы корректны и не являются скобками.
                if (firstArg !== null && second_arg !== null && first_arg !== '('
&& first_arg !== ')') {
                    // Генерируем результат узла и увеличиваем индекс.
                    nodeRes = `^${index.value++ - 1}`;
                    // Добавляем триаду с найденной операцией.
                    triads.push({
                        operation,
                        first_arg,
                        second_arg,
                        output: nodeRes,
                    });
                    // Обновляем `lastTriadOutput`.
                    lastTriadOutput = nodeRes;
                }
                // Возвращаем результат текущего узла.
```

```
                return nodeRes;
            }
            // Обрабатываем случай, когда лексема содержит минус.
            if (curNode.lexem.includes('-')) {
                // Проверяем наличие дочерних узлов.
                if (childNodeRes.length > 0) {
                    // Генерируем результат узла и увеличиваем индекс.
                    nodeRes = `^${index.value++ - 1}`;
                    // Определяем первый элемент дочернего результата.
                    const firstElem = childNodeRes[0];
                    // Если первый элемент существует, добавляем триаду для
минуса.
                    if (firstElem) {
                        triads.push({
                            operation: '-',
                            first_arg: lastTriadOutput,
                            second_arg: '--', // Специальный маркер для
отсутствующего второго аргумента.
                            output: nodeRes,
                        });
                        // Обновляем `lastTriadOutput`.
                        lastTriadOutput = nodeRes;
                        // Возвращаем результат текущего узла.
                        return nodeRes;
                    }
                }
            }
            // Возвращаем первый элемент дочернего результата.
            return childNodeRes[0];
        }
    }
    // Запускаем обработку корневого узла дерева.
    Tree_Processing(node);
    // Возвращаем массив сгенерированных триад.
    return triads;
  }
```

***Вспомогательный файл optimization.mjs***

```
// Функция для оптимизации триад.
export function Optimizator(triads) {
    // Создаем массив для хранения оптимизированных триад
    const finalOptimizedTriads = [];
    // Карта для хранения последних присваиваний для аргументов
    const prevAssign = new Map();
    // Множество для отслеживания уникальных операций
    const uniOp = new Set();
    // Карта для отслеживания новых выходных ссылок
    const outRefMap = new Map();
```

```
    // Массив для хранения удаленных строк (триад)
    const removded = [];
    // Переменная для хранения индекса первого присваивания

    // Переменные для хранения ссылок на проверочные операции
    let checking1 = 0;
    let checking2 = 0;
    // Переменные для проверки дубликатов операций
    let checking1_dublicate;
    let checking2_dublicate;
    // Счетчик для отслеживания количества операций
    let operationCounter = 0;
    // Флаг для обозначения первой операции
    let isFirstOperationFlag = 0;
    // Основной цикл по всем триадам
    for (const triad of triads) {
        // Проверяем, является ли операция присваиванием
        if (triad.operation === ':=') {
            // Получаем предыдущую триаду для текущего аргумента, если она
существует
            const PrevTriad = prevAssign.get(triad.first_arg);
            // Если есть предыдущая триада для текущего аргумента
            if (PrevTriad) {
                // Находим индекс этой триады в оптимизированных триадах
                const startIndex = finalOptimizedTriads.findIndex(t => t ===
PrevTriad);
                // Если индекс найден
                if (startIndex !== -1) {
                    // Проходим по триадам от начала до найденного индекса
                    for (let i = 0; i <= startIndex; i++) {
                        // Добавляем удаленные триады в массив removded
                        removded.push(finalOptimizedTriads[i]);
                        // Проверка для первой операции
                        if (isFirstOperationFlag < 1) {
                            isFirstOperationFlag++;
                            checking1 = `^${operationCounter}`;
                        }
                        operationCounter++;
                        checking2 = `${operationCounter}`;
                    }
                    // Удаляем все триады до и включая первую триаду присваивания
                    finalOptimizedTriads.splice(0, startIndex + 1);
                }
            }
            // Добавляем текущую триаду в оптимизированные триады
            finalOptimizedTriads.push(triad);
            // Сохраняем текущее присваивание для аргумента
            prevAssign.set(triad.first_arg, triad);
            // Сохраняем индекс первого присваивания
            firstAssignmentIndex = finalOptimizedTriads.length - 1;
            // Обновляем аргументы, если они совпадают с проверочной ссылкой
```

```
            if (triad.first_arg === `^${checking2}`) {
                triad.first_arg = checking1;
            }
            if (triad.second_arg === `^${checking2}`) {
                triad.second_arg = checking1;
            }
        } else {
            // Генерируем идентификатор для текущей триады (по операциям и
аргументам)
            const triadIdentifier = `${triad.operation}-${triad.first_arg}-
${triad.second_arg}`;
            // Если идентификатор триады не встречался, добавляем её
            if (!uniOp.has(triadIdentifier)) {
                finalOptimizedTriads.push(triad);
                uniOp.add(triadIdentifier);
                // Обновляем аргументы для проверки дубликатов
                if (triad.first_arg === checking1_dublicate) {
                    triad.first_arg = checking2_dublicate;
                } else if (triad.second_arg === checking1_dublicate) {
                    triad.second_arg = checking2_dublicate;
                }
            } else {
                // Если триада дублируется, сохраняем её для проверки и удаляем
                checking1_dublicate = triad.output;
                checking2_dublicate = finalOptimizedTriads.find(
                    t => t.operation === triad.operation &&
                         t.first_arg === triad.first_arg &&
                         t.second_arg === triad.second_arg
                )?.output;
                // Если триада совпадает с дубликатом, удаляем её
                if (triad.first_arg === checking1_dublicate && triad.operation
=== '-') {
                    const indexToRemove = finalOptimizedTriads.findIndex(t => t
=== triad);
                    if (indexToRemove !== -1) {
                        finalOptimizedTriads.splice(indexToRemove, 1);
                    }
                }
                // Обновляем аргументы триады, если они совпадают с проверочной
ссылкой
                if (triad.first_arg === checking1_dublicate) {
                    triad.first_arg = checking2_dublicate;
                } else if (triad.second_arg === checking1_dublicate) {
                    triad.second_arg = checking2_dublicate;
                }
            }
            // Обновление аргументов при совпадении с проверочной ссылкой
            if (triad.first_arg === `^${checking2}`) {
                triad.first_arg = checking1;
            }
            if (triad.second_arg === `^${checking2}`) {
```

```
                triad.second_arg = checking1;
            }
        }
    }
    // Создаем массив для отображения старых индексов триад на новые выходные
ссылки
    const newReferencesArr = finalOptimizedTriads.map((triad, index) => ({
        prevInd: index,
        newOutput: `^${index}`
    }));
    // Обновляем выходные ссылки триад, используя карту удаленных триад
    finalOptimizedTriads.forEach(triad => {
        removded.forEach(removed => {
            if (triad.output === removed.output) {
                const newIndex = newReferencesArr.find(mapping => mapping.prevInd
=== parseInt(removed.output.slice(1)));
                if (newIndex) {
                    triad.output = newIndex.newOutput;
                }
            }
        });
    });
    // Обновляем выходные ссылки триад и их аргументы на новые значения
    finalOptimizedTriads.forEach((triad, i) => {
        const newOutput = `^${i}`;
        outRefMap.set(triad.output, newOutput);
        triad.output = newOutput;

        if (outRefMap.has(triad.first_arg)) {
            triad.first_arg = outRefMap.get(triad.first_arg);
        }
        if (outRefMap.has(triad.second_arg)) {
            triad.second_arg = outRefMap.get(triad.second_arg);
        }
    });
    // Обновляем выходные ссылки в самих триадах с учетом их порядка
    for (let i = 0; i < finalOptimizedTriads.length; i++) {
        finalOptimizedTriads[i].output = `^${i}`;
    }
    // Возвращаем окончательно оптимизированные триады
    return finalOptimizedTriads;
}
```

***Вспомогательный файл assembler.mjs***

```
// Функция для перевода в ассемблер.
export function Assembler_Generator(triads) {
    // Создаём массив для хранения ассемблерных команд
    const assembly = [];
```

```
    // Счётчик для создания временных переменных (TMP0, TMP1 и т.д.)
    let tempVarCounter = 0;
    // Стек регистров для использования в ассемблерных командах
    const registerStack = ['AX'];
    // Индекс текущего регистра из стека
    let registerIndex = 0;
    // Словарь для хранения временных переменных, связанных с индексами триад
    const tempVars = {};
    // Проходим по всем триадам
    triads.forEach((triad, index) => {
        // Извлекаем операцию и аргументы триады
        const { operation, first_arg, second_arg } = triad;
        // Пропускаем триады с определёнными условиями (скобки и отсутствие
аргументов)
        if (first_arg === null || first_arg === '(' || second_arg === ')') {
            return; // Пропускаем, если триада некорректна
        }
        // Функция для разрешения аргументов (проверяет, являются ли они ссылками
на предыдущие триады)
        const resolveArgument = (arg) => {
            // Если аргумент — строка, начинающаяся с '^', извлекаем значение из
`tempVars` или возвращаем сам аргумент
            if (typeof arg === 'string' && arg.startsWith('^')) {
                const refIndex = parseInt(arg.slice(1), 10); // Преобразуем
ссылку в индекс
                return tempVars[refIndex] || arg; // Возвращаем значение или саму
ссылку
            }
            // Возвращаем аргумент, если это не ссылка
            return arg;
        };
        // Разрешаем значения первого и второго аргумента
        const first_argValue = resolveArgument(first_arg);
        const second_argValue = resolveArgument(second_arg);
        // Текущий регистр для использования в командах
        let targetRegister = registerStack[registerIndex];
        // Создаём имя для новой временной переменной
        let tempVar = `TMP${tempVarCounter++}`;
        // Обрабатываем триаду в зависимости от типа операции
        switch (operation) {
            case ':=':
                // Генерируем команду присваивания (MOV)
                assembly.push({ command: `MOV ${first_argValue},
${targetRegister}` });
                // Сохраняем переменную, связав её с текущей триадой
                tempVars[index] = first_argValue;
                break;

            case '+':
                // Генерируем команды для сложения (MOV и ADD)
```

```
                assembly.push({ command: `MOV ${targetRegister},
${first_argValue}` });
                assembly.push({ command: `ADD ${targetRegister},
${second_argValue}` });
                assembly.push({ command: `MOV ${tempVar}, ${targetRegister}` });
                // Сохраняем временную переменную
                tempVars[index] = tempVar;
                break;

            case '*':
                // Генерируем команды для умножения (MUL)
                assembly.push({ command: `MUL ${second_argValue}` });
                assembly.push({ command: `MOV ${tempVar}, ${targetRegister}` });
                // Сохраняем временную переменную
                tempVars[index] = tempVar;
                break;

            case '/':
                // Генерируем команды для деления (MOV и DIV)
                assembly.push({ command: `MOV ${targetRegister},
${first_argValue}` });
                assembly.push({ command: `DIV ${second_argValue}` });
                assembly.push({ command: `MOV ${tempVar}, ${targetRegister}` });
                // Сохраняем временную переменную
                tempVars[index] = tempVar;
                break;

            case '-':
                // Генерируем команды для вычитания (NEG)
                assembly.push({ command: `NEG ${targetRegister}` });
                assembly.push({ command: `MOV ${tempVar}, ${targetRegister}` });
                // Сохраняем временную переменную
                tempVars[index] = tempVar;
                break;

            default:
                // Логирование ошибки при неопределённой операции
                console.error(`Error (undefined operation): ${operation}`);
        }
    });
    // Возвращаем массив ассемблерных команд
    return assembly;
}
```

**Примеры работы программы**

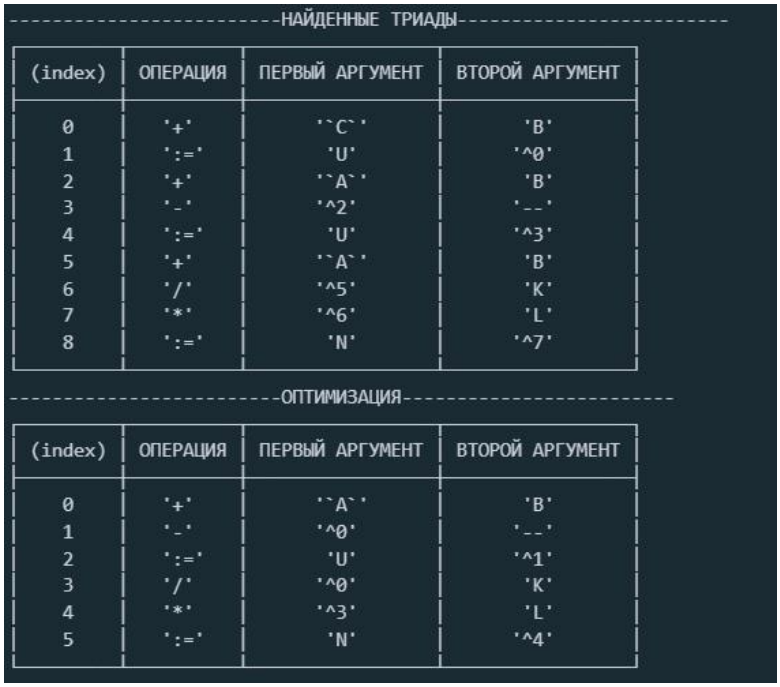
Рассмотрим первый пример, который представлен ниже:

$$U \coloneqq C + B;$$

$$U \coloneqq -(`A` + B);$$

$$N \coloneqq (`A` + B)/K * L;$$

Как видно из представленного примера, он в себе содержит, во-первых, повторяющееся выражение $(`A` + B)$, только во второй строке оно идёт с отрицанием, а третья строка использует его без него. Результатом работы программы ожидается такое поведение, что в результате проделанных операций (в частности, после оптимизации) программа, увидевшая эту повторяющуюся операцию, оптимизирует третью строку так, что вместо $(`A` + B)$ будет использоваться ссылка на такую же структуру из второй строки (до операции отрицания). Во-вторых, здесь также можно заметить повторное присваивание переменной U. Здесь ожидается, что программа определит в моменте оптимизации, что присваивание переменной U повторяется, отчего первое присваивание лишается смысла. Поэтом результатом оптимизации данного момента является отсутствие триад, связанных с первым присваиванием. Оптимизированные триады должны будут начинаться со второй строки. Результаты продемонстрированы на рисунках 3 – 4.



НАЙДЕННЫЕ ТРИАДЫ

| (index) | ОПЕРАЦИЯ | ПЕРВЫЙ АРГУМЕНТ | ВТОРОЙ АРГУМЕНТ |
|---|---|---|---|
| 0 | '+' | '`C`' | 'B' |
| 1 | ':=' | 'U' | '^0' |
| 2 | '+' | '`A`' | 'B' |
| 3 | '-' | '^2' | '--' |
| 4 | ':=' | 'U' | '^3' |
| 5 | '+' | '`A`' | 'B' |
| 6 | '/' | '^5' | 'K' |
| 7 | '*' | '^6' | 'L' |
| 8 | ':=' | 'N' | '^7' |

ОПТИМИЗАЦИЯ

| (index) | ОПЕРАЦИЯ | ПЕРВЫЙ АРГУМЕНТ | ВТОРОЙ АРГУМЕНТ |
|---|---|---|---|
| 0 | '+' | '`A`' | 'B' |
| 1 | '-' | '^0' | '--' |
| 2 | ':=' | 'U' | '^1' |
| 3 | '/' | '^0' | 'K' |
| 4 | '*' | '^3' | 'L' |
| 5 | ':=' | 'N' | '^4' |

Рисунок 3 – Результат работы программы

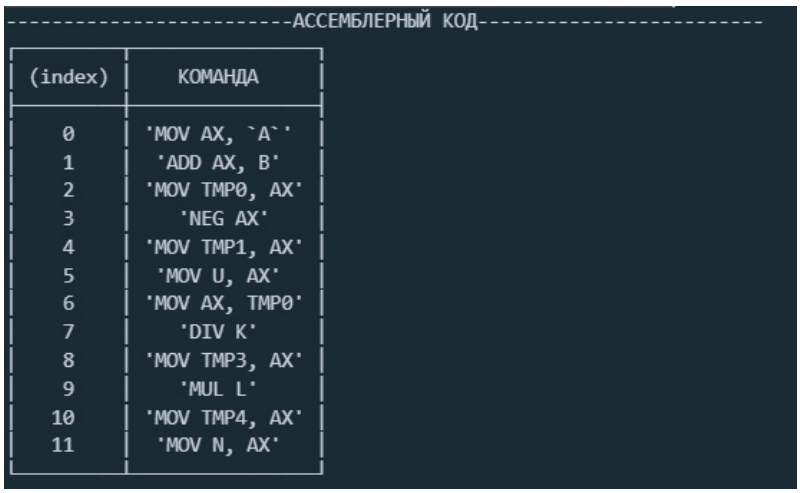



Рисунок 4 – Результат работы программы

Рассмотрим рисунок 4 внимательнее. В момент генерации ассемблерного кода предполагалось использование временных переменных вида ***TMPi***, которые хранили бы в себе результат операции. Это сделано для того, чтобы избавиться от ссылок в качестве аргументов. Теперь, с использованием этих временных переменных вместо ссылки будет указываться временная переменная, как это и видно на рисунке 4.

Следует также отметить, что программа запоминает действия во временные переменные по типу сложения, деления, умножения, вычитания на случай повторного использования.

Разберём второй пример. Он имеет вид, как представлено ниже:

$$N \coloneqq (`X` + B) * H/P;$$

$$N \coloneqq C + `A`;$$

$$U \coloneqq `Q`;$$

Результатом работы программы также ожидается удаление триад, связанных с первой строкой, потому что здесь используется повторное присваивание. Результаты продемонстрированы на рисунках 5 – 6.

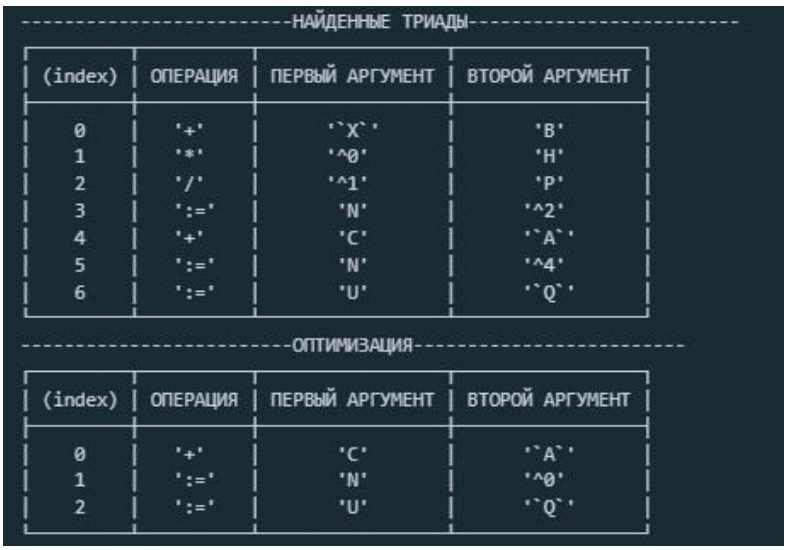



Рисунок 5 – Результат работы программы

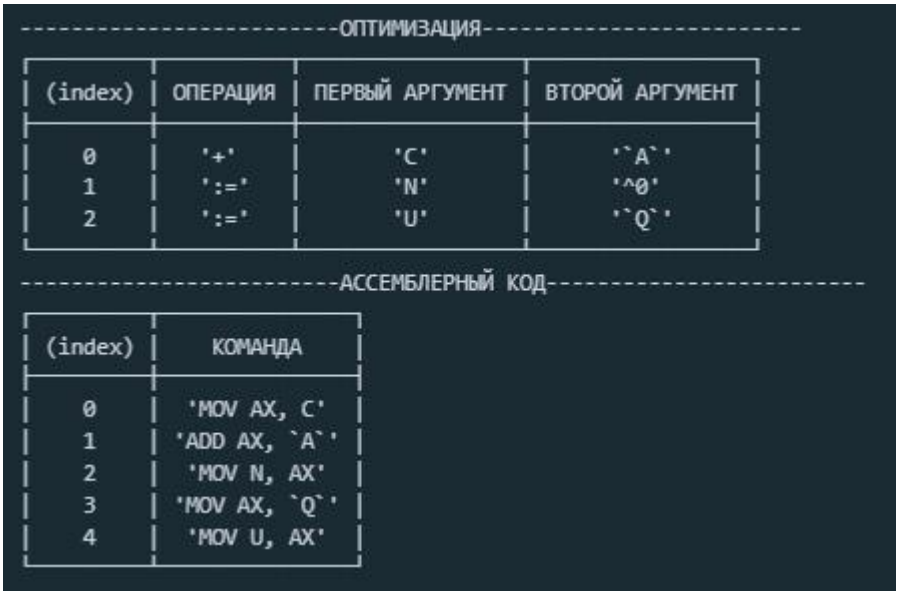



Рисунок 6 – Результат работы программы

**Выводы**

В данной лабораторной работы были изучены основные принципы генерации компилятором объектного кода, выполнена генерация объектного кода программы на основе результатов синтаксического анализа для заданного входного языка.

Изучены основные принципы оптимизации компилятором объектного кода для линейного участка программы, прошло ознакомление с методами оптимизации результирующего объектного кода с помощью метода исключения лишних операций.