МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ РОССИЙСКОЙ ФЕДЕРАЦИИ
федеральное государственное автономное образовательное учреждение высшего образования
«САНКТ-ПЕТЕРБУРГСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ
АЭРОКОСМИЧЕСКОГО ПРИБОРОСТРОЕНИЯ»

КАФЕДРА №44

КУРСОВАЯ РАБОТА (ПРОЕКТ)
ЗАЩИЩЕНА С ОЦЕНКОЙ

РУКОВОДИТЕЛЬ

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 дисциплине: Системное программное обеспечение

РАБОТУ ВЫПОЛНИЛ

СТУДЕНТ гр. № 4142		Д.Р. Рябов
	подпись, дата	инициалы, фамилия

Санкт-Петербург 2024

1. **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 методов свертки объектного кода и исключения лишних операций.

2. Задание:

Вариант 19. Грамматика 1. В список допустимых лексем входят: Идентификаторы, символьные константы

Написать и отладить программу генерации и оптимизации программного кода на основе результатов синтаксического анализа заданной входной цепочки в формате ассемблерных команд. Текст на входном языке задается в виде символьного (текстового) файла.

3. Запись заданной грамматики входного языка в форме Бэкуса-Наура

Язык G({S, F, T, E}, {:=,-,+,*,/,(,), 1, a, ‘a’}, P, S):

S ->a := F;

F-> F+T | T

T-> T*E | T/E | E

E-> (F) | -(F) | a

4. Список возможных триад с расшифровкой.

Все шифры команд были заимствованы из intel 8086:

- mov – команда присваивания,
- mul – умножение,
- add - сложение,
- div – деление,
- sub – вычитание,

5. Разработка программы

Будем использовать как входные данные результат выполнения программы из прошлого семестра (считаем из json файла)

Код программы представлен далее:

Main.go:

```
package main


import (
   "encoding/json"
   "fmt"
   "lab1_2/code_generation"
   "lab1_2/node"
   "lab1_2/optimizer"
   "lab1_2/triad"
   "log"
   "os"
)


func main() {
   var nodes []node.Node
   // читаем файл, сгенерированный алгоритмом из прошлых ЛР
   fileData, err := os.ReadFile("../output.json")
   if err != nil {
      log.Fatalf("Ошибка при чтении файла: %v", err)
   }


   if err := json.Unmarshal(fileData, &nodes); err != nil {
      fmt.Println("Ошибка при декодировании JSON:", err)
      return
   }
   println("Начальные выражения:")
   for _, node := range nodes {
      // Вывод начальной лексеммы
      fmt.Printf("%+v\n", node.Lexem)
   }


   var doubleTriads [][]triad.Triad


   println("Триады:")
   for _, node := range nodes {
      // Вывод начальной лексеммы
      var triads []triad.Triad
      triad.ConvertNodeToTriads(node, &triads)
      // Печать триад
```

```
        doubleTriads = append(doubleTriads, triads)
    }


    resultTriads := triad.MergeTriadList(doubleTriads...)
    printTriads(resultTriads)
    println("Код до оптимизации:")
    res := code_generation.GenerateAssemblyCode(resultTriads)
    println(res)


    println("триады после оптимизации:")
    optimizer.OptimizeTriads(&resultTriads)
    printTriads(resultTriads)


    println("Код после оптимизации:")
    res = code_generation.GenerateAssemblyCode(resultTriads)
    println(res)


}


func printTriads(triads []triad.Triad) {
    for i, t := range triads {
        fmt.Printf("%d: %s\n", i+1, t.ToString())
    }
}
```

Types.go:

```
package types


const (
    Delimiter     = ","                                          // Символ разделителя
    Alphabet      = "abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ"          // Алфавит
    Alphanumeric  = "abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ0123456789" //
Алфавит и цифры
    Numbers       = "0123456789"                                 // Цифры
    OperatorChars = "+-*/"                                       // Символы операторов
    Parentheses   = "()"                                         // Символы скобок
    Identifier    = ":="
)
```

```
const (
    Terminal    = "Terminal"
    NonTerminal = "NonTerminal"
)
```

node.go:

```
package node


import (
    "lab1_2/types"
    "strings"
)


type Node struct {
    Type     string `json:"type"`     // Тип узла (Terminal или NonTerminal)
    Lexem    string `json:"lexem"`    // Лексема (строка, представляющая символ)
    Children []Node `json:"children"` // Дочерние элементы (если есть)
}


// ContainsNonTerminal проверяет, содержит ли нода хотя бы один дочерний нода типа
NonTerminal
func (node Node) ContainsNonTerminal() bool {
    for _, child := range node.Children {
        if !child.IsTerminal() {
            return true
        }
    }
    return false
}


func (node Node) IsTerminal() bool {
    return node.Type == types.Terminal
}


// IsSimple проверяет, является ли нода "простым" (терминалом или узлом с простыми
лексемами)
func (node Node) IsSimple() bool {
    if len(node.Children) == 0 {
        return true
    }
    for _, child := range node.Children {
        if child.ContainsNonTerminal() || !child.IsTerminal() {
```

```
            return false
        }
    }
    return true
}


// IsSimpleArithmetic проверяет, является ли нода арифметическим выражением (например, "a + b")
func (node Node) IsSimpleArithmetic() bool {
    if len(node.Children) == 3 && !node.ContainsNonTerminal() {
        // Для арифметического выражения должно быть 3 элемента: переменная, оператор и переменная
        return strings.ContainsAny(node.Children[1].Lexem, types.OperatorChars)
    }
    return false
}


func (node Node) HasDelimeter() bool {
    for _, child := range node.Children {
        if child.HasDelimeter() || child.Lexem == types.Delimiter {
            return true
        }
    }
    return false
}


// IsAssignment проверяет, является ли нода присваиванием (например, "a := 3")
func (node Node) IsAssignment() bool {
    if len(node.Children) == 3 {
        return node.Children[1].Lexem == types.Identifier
    }
    return false
}


// IsParenthesesExpression проверяет, является ли выражение в скобках (например, "(a + b)")
func (node Node) IsParenthesesExpression() bool {
    if len(node.Children) >= 3 {
        return node.Children[0].Lexem == "(" || node.Children[2].Lexem == ")"
    }
    return false
}


// IsComplexExpression проверяет, является ли нода сложным выражением с несколькими операторами
func (node Node) IsComplexExpression() bool {
    if node.Type == types.NonTerminal {
```

```
        return true
    }
    // Проверка на наличие арифметических выражений в дочерних узлах
    return node.ContainsNonTerminal()
}


// ContainsLink проверяет, ссылается ли нода на другие узлы
func (node Node) ContainsLink() bool {
    for _, child := range node.Children {
        if child.Type == types.NonTerminal {
            return true
        }
    }
    return false
}


// PrintLexems выводит лексемы всех узлов в дереве
func (node Node) PrintLexems() {
    if node.Type == types.Terminal {
        print(node.Lexem + " ")
    }
    for _, child := range node.Children {
        child.PrintLexems()
    }
}
```

operand.go:

```
package triad


import (
    "fmt"
    "lab1_2/node"
    "lab1_2/types"
    "strconv"
    "strings"
)


// Operand представляет операнд триады.
// Это может быть:
// - Простое значение (например, число или переменная), хранящееся в поле `element`
```

```go
// - Ссылка на другую триаду, хранящаяся в поле `linkTo`
type Operand struct {
    element string // Значение операнда (например, число, переменная или строка)
    linkTo  *int   // Указатель на индекс триады, на которую ссылается данный операнд
}


// GetOperand возвращает строковое представление операнда.
// Если это ссылка, то возвращается "^<индекс>".
// Если это значение, то возвращается само значение.
func (o Operand) GetOperand() string {
    if o.IsLink() {
        return fmt.Sprintf("^%v", *o.linkTo) // Формат ссылки
    }
    return o.element // Простое значение
}


// IsLink проверяет, является ли операнд ссылкой на другую триаду.
func (o Operand) IsLink() bool {
    return o.linkTo != nil
}


// IsNumber проверяет, является ли операнд числом.
// Использует набор символов, определённый в `types.Numbers`.
func (o Operand) IsNumber() bool {
    return strings.ContainsAny(o.element, types.Numbers)
}


// IsVariable проверяет, является ли операнд переменной.
// Использует набор символов, определённый в `types.Alphabet`.
func (o Operand) IsVariable() bool {
    return strings.ContainsAny(o.element, types.Alphabet)
}


// GetLink возвращает указатель на индекс ссылки операнда.
// Если операнд не является ссылкой, возвращает `nil`.
func (o Operand) GetLink() *int {
    return o.linkTo
}


// SetLink устанавливает ссылку на указанную триаду.
func (o Operand) SetLink(link int) {
    o.linkTo = &link
}
```

```go
// OperandFromString создаёт операнд из строки.
// Пример: "x" -> Operand{element: "x"}
func OperandFromString(s string) Operand {
    return Operand{
        element: s,
    }
}


// NumberOperand создаёт операнд из числа.
// Число преобразуется в строку для хранения в `element`.
// Пример: 42 -> Operand{element: "42"}
func NumberOperand(n int) Operand {
    return Operand{
        element: strconv.Itoa(n),
    }
}


// OperandFromSimpleNode создаёт операнд из узла (Node).
// Использует поле `Lexem` узла как значение операнда.
// Пример: Node{Lexem: "x"} -> Operand{element: "x"}
func OperandFromSimpleNode(n node.Node) Operand {
    return Operand{
        element: n.Lexem,
    }
}


// LinkOperand создаёт операнд-ссылку на указанную триаду.
// Пример: 3 -> Operand{linkTo: &3}
func LinkOperand(index int) Operand {
    return Operand{
        linkTo: &index,
    }
}
```

Triad.go:

```go
package triad


import "fmt"


// Triad представляет триаду, которая состоит из:
// - Оператора (например, "+", "-", "*" и т.д.)
```

```go
// - Двух операндов (Operand1 и Operand2), которые являются входными значениями для операции
type Triad struct {
    Operator string  // Оператор, выполняющий операцию
    Operand1 Operand // Первый операнд (входное значение 1)
    Operand2 Operand // Второй операнд (входное значение 2)
}


// ToString возвращает строковое представление триады в формате:
// "Оператор(Операнд1, Операнд2)"
func (t Triad) ToString() string {
    return fmt.Sprintf("%s(%s, %s)", t.Operator, t.Operand1.GetOperand(), t.Operand2.GetOperand())
}


// Equals проверяет равенство двух триад. Две триады считаются равными, если:
// - Их операторы совпадают
// - Оба операнда совпадают по значениям
// Не обойтись простым сравнением структур, поскольку в операндах используются ссылки
// и при сравнении они могут ссылаться на разные адреса в памяти
func (t Triad) Equals(tr Triad) bool {
    return t.Operand1.GetOperand() == tr.Operand1.GetOperand() &&
        t.Operand2.GetOperand() == tr.Operand2.GetOperand() &&
        t.Operator == tr.Operator
}


// MergeTriadList объединяет несколько списков триад и корректирует ссылки
func MergeTriadList(triadsList ...[]Triad) []Triad {
    var outputTriads []Triad
    offset := 0 // Смещение для обновления ссылок


    for _, triads := range triadsList {
        for _, triad := range triads {
            // Копируем триаду, чтобы избежать изменения исходного списка
            newTriad := triad


            // Обновляем ссылки, если они есть
            if newTriad.Operand1.IsLink() {
                newIndex := *newTriad.Operand1.linkTo + offset
                newTriad.Operand1 = LinkOperand(newIndex)
            }
            if newTriad.Operand2.IsLink() {
                newIndex := *newTriad.Operand2.linkTo + offset
                newTriad.Operand2 = LinkOperand(newIndex)
            }
```

```
            // Добавляем обновленную триаду в результирующий список
            outputTriads = append(outputTriads, newTriad)
        }
        // Обновляем смещение на длину текущего списка триад
        offset += len(triads)
    }


    return outputTriads
}
```

Triad_converter.go:

```go
package triad


import (
    "lab1_2/node"
    "lab1_2/types"
    "strings"
)


// ConvertNodeToTriads конвертирует узел (node) в триады и возвращает последний операнд.
// Аргументы:
// - nodeToConvert: узел для преобразования
// - triads: указатель на список, куда будут добавлены новые триады
func ConvertNodeToTriads(nodeToConvert node.Node, triads *[]Triad) Operand {
    var lastOperator Operand
    switch {
    case nodeToConvert.HasDelimeter():
        // Если узел содержит разделитель, конвертируем его дочерний узел
        ConvertNodeToTriads(nodeToConvert.Children[0], triads)
    case nodeToConvert.IsParenthesesExpression():
        // Если узел является выражением в скобках, обрабатываем внутреннее выражение
        innerNode := nodeToConvert.Children[1] // Внутреннее выражение в скобках
        lastOperator = ConvertNodeToTriads(innerNode, triads)
    case nodeToConvert.IsSimpleArithmetic():
        // Если узел является простым арифметическим выражением, конвертируем его в триаду
        ConvertArithmeticNodeToTriad(nodeToConvert, triads)
    case nodeToConvert.IsComplexExpression() || nodeToConvert.IsAssignment():
        // Если узел является сложным выражением или присваиванием, конвертируем его в
триаду
        ConvertExpressionToTriad(nodeToConvert, triads)
    default:
        // Для всех остальных случаев рекурсивно обрабатываем дочерние узлы
        for _, child := range nodeToConvert.Children {
            lastOperator = ConvertNodeToTriads(child, triads)
        }
    }
    // Если последний оператор пустой, создаем ссылку на текущую длину списка триад
    if lastOperator.GetOperand() == "" {
        lastOperator = LinkOperand(len(*triads))
    }


    return lastOperator
}
```

```go
// ConvertArithmeticNodeToTriad преобразует узел с арифметическим выражением в триаду
// Аргументы:
// - nodeToConvert: узел для преобразования
// - triads: указатель на список триад
// Возвращает:
// - Новую триаду
func ConvertArithmeticNodeToTriad(nodeToConvert node.Node, triads *[]Triad) Triad {
    operator := nodeToConvert.Children[1].Lexem                // Оператор (например, "+", "-")
    operand := OperandFromSimpleNode(nodeToConvert.Children[0])   // Первый операнд
    operator2 := OperandFromSimpleNode(nodeToConvert.Children[2]) // Второй операнд


    // Создаем новую триаду
    newTriad := Triad{
        Operator: operator,
        Operand1: operand,
        Operand2: operator2,
    }


    // Добавляем триаду в список
    *triads = append(*triads, newTriad)
    return newTriad
}


// ConvertExpressionToTriad преобразует сложное выражение или присваивание в триаду
// Аргументы:
// - nodeToConvert: узел для преобразования
// - triads: указатель на список триад
// Возвращает:
// - Новую триаду
func ConvertExpressionToTriad(nodeToConvert node.Node, triads *[]Triad) Triad {
    var operand1, operand2 Operand
    var operator string


    // Конвертируем первый и второй операнды
    operand1 = ConvertToOperand(nodeToConvert.Children[0], triads)
    operand2 = ConvertToOperand(nodeToConvert.Children[2], triads)


    // Определяем оператор
    if strings.ContainsAny(nodeToConvert.Children[1].Lexem, types.OperatorChars) {
        operator = nodeToConvert.Children[1].Lexem
    } else if nodeToConvert.Children[1].Lexem == types.Identifier {
        operator = types.Identifier
    }


    // Создаем и добавляем новую триаду
```

```
    newTriad := Triad{
        Operator: operator,
        Operand1: operand1,
        Operand2: operand2,
    }


    *triads = append(*triads, newTriad)
    return newTriad
}


func ConvertToOperand(n node.Node, triads *[]Triad) Operand {
    var o Operand
    if n.IsSimple() && n.IsTerminal() {
        o = OperandFromSimpleNode(n)
    } else {
        o = ConvertNodeToTriads(n, triads)
    }
    return o
}
```

Optimizer.go:

```
package optimizer


import (
    "lab1_2/triad"
)


// Мапа для хранения значений переменных, которые стали известны
var constantsTable map[string]int = make(map[string]int)


// OptimizeTriads выполняет свертку триад
func OptimizeTriads(triads *[]triad.Triad) {
    // Проход по всем триадам
    for index, triad := range *triads {
        // Пробуем свертку для текущей триады
        countValueIfPossible(triad, index, triads)
    }
    // удаляем триады с константами
    removeRedundantTriadsWithConstants(triads)
```

```
    // определяем same триады
    *triads = eliminateRedundantOperations(*triads)
    // удяляем same триады
    removeSameTriads(triads)
}
```

Constants.go:

```
package optimizer


import (
    "fmt"
    "lab1_2/triad"
    "lab1_2/types"
    "strconv"
)


// countValueIfPossible пытается свернуть триаду, если это возможно
func countValueIfPossible(t triad.Triad, index int, triads *[]triad.Triad) {
    // Шаг 1: Если операнд1 является ссылкой, заменяем её на константу
    if t.Operand1.IsLink() {
        changeLinkOperand(&t.Operand1, *triads)
    }


    // Шаг 2: Если операнд2 является ссылкой, заменяем её на константу
    if t.Operand2.IsLink() {
        changeLinkOperand(&t.Operand2, *triads)
    }


    // Шаг 3: Пытаемся заменить переменную на константу, если такая есть в таблице констант
    tryReplaceVariableWithConstant(&t.Operand1, &t.Operand2, t.Operator, constantsTable)
    tryReplaceVariableWithConstant(&t.Operand2, &t.Operand1, t.Operator, constantsTable)


    // Шаг 4: Если операнд1 — переменная, а операнд2 — число, и оператор присваивания,
добавляем значение в таблицу констант
    if t.Operand1.IsVariable() && t.Operand2.IsNumber() && t.Operator == types.Identifier {
        num, err := strconv.Atoi(t.Operand2.GetOperand())
        if err != nil {
            fmt.Printf("Ошибка преобразования в число: %e", err)
        }
        constantsTable[t.Operand1.GetOperand()] = num
    }
```

```
    // Шаг 5: Если оба операнда — числа, то выполняем операцию и заменяем триаду на
"константную" (C(K, 0))
    if t.Operand1.IsNumber() && t.Operand2.IsNumber() {
        result := performOperation(t) // Выполняем арифметическую операцию
        t = triad.Triad{
            Operator: "C",
            Operand1: triad.OperandFromString(fmt.Sprintf("%d", result)),
            Operand2: triad.OperandFromString("0"),
        }
    }
    (*triads)[index] = t // Обновляем триаду в исходном списке
}


// changeLinkOperand заменяет ссылочный операнд на константу, если это возможно
func changeLinkOperand(o *triad.Operand, triads []triad.Triad) {
    linkIndex := *o.GetLink()         // Получаем индекс ссылки
    linkedTriad := triads[linkIndex-1] // Находим связанную триаду
    if linkedTriad.Operator == "C" && linkedTriad.Operand2.GetOperand() == "0" {
        // Если триада содержит константу, заменяем операнд на её значение
        *o = triad.OperandFromString(linkedTriad.Operand1.GetOperand())
    }
}


// tryReplaceVariableWithConstant пытается заменить переменную на константу из таблицы
констант
func tryReplaceVariableWithConstant(operand *triad.Operand, otherOperand *triad.Operand,
operator string, constantsTable map[string]int) {
    // Если операнд — переменная, заменяем её на константу, если она есть в таблице
    if operand.IsVariable() && (!otherOperand.IsNumber() || operator != types.Identifier) {
        if value, exists := constantsTable[operand.GetOperand()]; exists {
            *operand = triad.NumberOperand(value) // Замена переменной на её значение
        }
    }
}


// performOperation выполняет арифметическую операцию над двумя числовыми операндами
func performOperation(triad triad.Triad) int {
    var operand1, operand2 int
    // Преобразуем строковые значения операндов в числа
    fmt.Sscanf(triad.Operand1.GetOperand(), "%d", &operand1)
    fmt.Sscanf(triad.Operand2.GetOperand(), "%d", &operand2)


    // Выбираем операцию в зависимости от оператора
    switch triad.Operator {
    case "+":
```

```
        return operand1 + operand2
    case "*":
        return operand1 * operand2
    case "-":
        return operand1 - operand2
    case "/":
        return operand1 / operand2
    }
    return 0
}


func removeRedundantTriadsWithConstants(triads *[]triad.Triad) {
    var optimizedTriads []triad.Triad
    for _, t := range *triads {
        // Проверяем, является ли триада C(K, 0)
        if t.Operator == "C" && t.Operand2.GetOperand() == "0" {
            // Пропускаем эту триаду (не добавляем в результат)
            continue
        }
        // Добавляем триаду в оптимизированный список
        optimizedTriads = append(optimizedTriads, t)
    }
    // Обновляем исходный список триад
    *triads = optimizedTriads
}
```

Remove_same.go:

```
package optimizer


import (
    "lab1_2/triad"
)


var depTriads []int // хранит зависимости для каждой триады
// Основной алгоритм исключения лишних операций
func eliminateRedundantOperations(triads []triad.Triad) []triad.Triad {
    dep := make(map[string]int) // хранит зависимости переменных
    result := []triad.Triad{}   // итоговый результат
```

```
depTriads = make([]int, len(triads))


for i, t := range triads {
    // Шаг 1: Замена операндов, ссылающихся на SAME
    if t.Operand1.IsLink() {
        linkedTriad := triads[*t.Operand1.GetLink()-1]
        if linkedTriad.Operator == "SAME" {
            // Проверяем, есть ли уже ссылка в операнде
            if t.Operand1.IsLink() {
                // Если ссылка уже есть, то мы просто заменяем ее на нужную
                t.Operand1.SetLink(*linkedTriad.Operand1.GetLink())
            } else {
                // Если ссылки нет, то мы создаем новую ссылку
                t.Operand1 = triad.LinkOperand(*linkedTriad.Operand1.GetLink())
            }
        }
    }
    if t.Operand1.IsLink() {
        linkedTriad := triads[*t.Operand2.GetLink()-1]
        if linkedTriad.Operator == "SAME" {
            // Проверяем, есть ли уже ссылка в операнде
            if t.Operand2.IsLink() {
                // Если ссылка уже есть, то мы просто заменяем ее на нужную
                t.Operand2.SetLink(*linkedTriad.Operand2.GetLink())
            } else {
                // Если ссылки нет, то мы создаем новую ссылку
                t.Operand2 = triad.LinkOperand(*linkedTriad.Operand2.GetLink())
            }
        }
    }


    // Шаг 2: Вычисление числа зависимости текущей триады
    depTriads[i] = 1 + max(calcDependency(t.Operand1, dep), calcDependency(t.Operand2, dep))


    // Шаг 3: Проверка на идентичность с более ранней триадой
    redundant, j := checkIfRedundant(t, triads, i)
    if redundant {
        result = append(result, triad.Triad{
            Operator: "SAME",
            Operand1: triad.LinkOperand(j + 1), // Ссылка на триаду с номером i
            Operand2: triad.NumberOperand(0),
        })
    } else {
        result = append(result, t)


    }
```

```
        // Шаг 4: Присваивание числа зависимости переменным
        if t.Operator == ":=" {
            dep[t.Operand1.GetOperand()] = i
        }
    }

    return result
}

// Функция для расчета числа зависимости для операндов
func calcDependency(operand triad.Operand, dep map[string]int) int {
    if operand.IsVariable() {
        // возвращаем зависимость переменной
        return dep[operand.GetOperand()]
    }
    // Для констант или значений без зависимости возвращаем 0
    return 0
}

func checkIfRedundant(t triad.Triad, triads []triad.Triad, index int) (bool, int) {
    for j, tr := range triads {
        if depTriads[index] == depTriads[j] && t.Equals(tr) && j < index {
            return true, j
        }
    }
    return false, 0
}

// Вспомогательная функция для поиска максимума
func max(a, b int) int {
    if a > b {
        return a
    }
    return b
}

// Функция для удаления всех SAME триад и обновления ссылок
func removeSameTriads(triads *[]triad.Triad) {
    var result []triad.Triad
    // Мапа для отслеживания замененных ссылок
    // Ключ - это индекс операнда, который ссылается на SAME триаду, а значение - индекс, на
который нужно обновить ссылку
    linkUpdates := make(map[int]int)
```

```
    // Проходим по всем триадам
    for i, t := range *triads {
        if t.Operator == "SAME" {
            // Если это SAME триада, то мы обновляем ссылку для всех последующих триад, которые на нее ссылаются
            if t.Operand1.IsLink() {
                // Получаем индекс ссылки на триаду SAME
                linkedIndex := *t.Operand1.GetLink()
                // Обновляем ссылку на настоящую триаду
                linkUpdates[i+1] = linkedIndex
            }
            // Пропускаем SAME триаду, так как она не должна быть в результате
            continue
        }

        // Проверяем, если операнды ссылаются на удаленную SAME триаду, заменяем ссылки
        if t.Operand1.IsLink() {
            linkedIndex := *t.Operand1.GetLink()
            if newLink, exists := linkUpdates[linkedIndex]; exists {
                // Обновляем ссылку на новую триаду
                t.Operand1 = triad.LinkOperand(newLink)
            }
        }

        if t.Operand2.IsLink() {
            linkedIndex := *t.Operand2.GetLink()
            if newLink, exists := linkUpdates[linkedIndex]; exists {
                // Обновляем ссылку на новую триаду
                t.Operand2 = triad.LinkOperand(newLink)
            }
        }

        if redundant, j := checkIfRedundant(t, result, i); !redundant {
            // Добавляем триаду в итоговый результат
            result = append(result, t)
            depTriads[len(result)-1] = depTriads[i]

            // Обновляем ссылки для всех триад, ссылающихся на эту
            linkUpdates[i+1] = len(result)
        } else {
            linkUpdates[i+1] = j + 1
        }

    }
```

```
    *triads = result
}
```

Code_generator.go:

```
package code_generation


import (
    "fmt"
    "lab1_2/triad"
    "strings"
)


var updatedLinkOperands map[int]string = make(map[int]string)


// GenerateAssemblyCode генерирует ассемблерный код на основе списка триад
func GenerateAssemblyCode(triads []triad.Triad) string {
    var assemblyCode strings.Builder


    // Переменная для отслеживания временных регистров (например, AX, BX, CX, DX)
    regIndex := 0


    // Обрабатываем каждую триаду
    for i, triad := range triads {
        operand1 := operandToString(triad.Operand1, i, triads)
        operand2 := operandToString(triad.Operand2, i, triads)
        switch triad.Operator {
        case "*":
            // Умножение: *(B,C)
            assemblyCode.WriteString(fmt.Sprintf("mul %s, %s\n", operand1, operand2))
            regIndex++ // Увеличиваем индекс для следующей операции


        case "+":
            // Сложение: +(операнд1, операнд2)
            assemblyCode.WriteString(fmt.Sprintf("add %s,%s\n", operand1, operand2))
            regIndex++ // Увеличиваем индекс для следующей операции


        case "-":
```

```
            // Вычитание: -(операнд1, операнд2)
            assemblyCode.WriteString(fmt.Sprintf("sub %s,%s\n", operand1, operand2))
            regIndex++ // Увеличиваем индекс для следующей операции


        case "/":
            // деление: /(операнд1, операнд2)
            assemblyCode.WriteString(fmt.Sprintf("div %s,%s\n", operand1, operand2))
            regIndex++ // Увеличиваем индекс для следующей операции


        case ":=":
            // Присваивание: :=(A, операнд)
            assemblyCode.WriteString(fmt.Sprintf("mov %s, %s\n", operand1, operand2))


            // Сохраняем результат присваивания
            updatedLinkOperands[i] = operand1
        default:
            assemblyCode.WriteString(fmt.Sprintf("Unknown operator: %s\n", triad.Operator))
        }
    }


    return assemblyCode.String()
}


func operandToString(o triad.Operand, index int, triads []triad.Triad) string {
    if o.IsLink() {
        // Проверяем, есть ли результат в `updatedLinkOperands`
        if linkedOperand, ok := updatedLinkOperands[*o.GetLink()]; ok {
            return linkedOperand
        }


        // Если результат не найден, возвращаем операнд из ссылки
        return triads[*o.GetLink()-1].Operand1.GetOperand()
    }


    return o.GetOperand()
}
```

Результат выполнения программы:

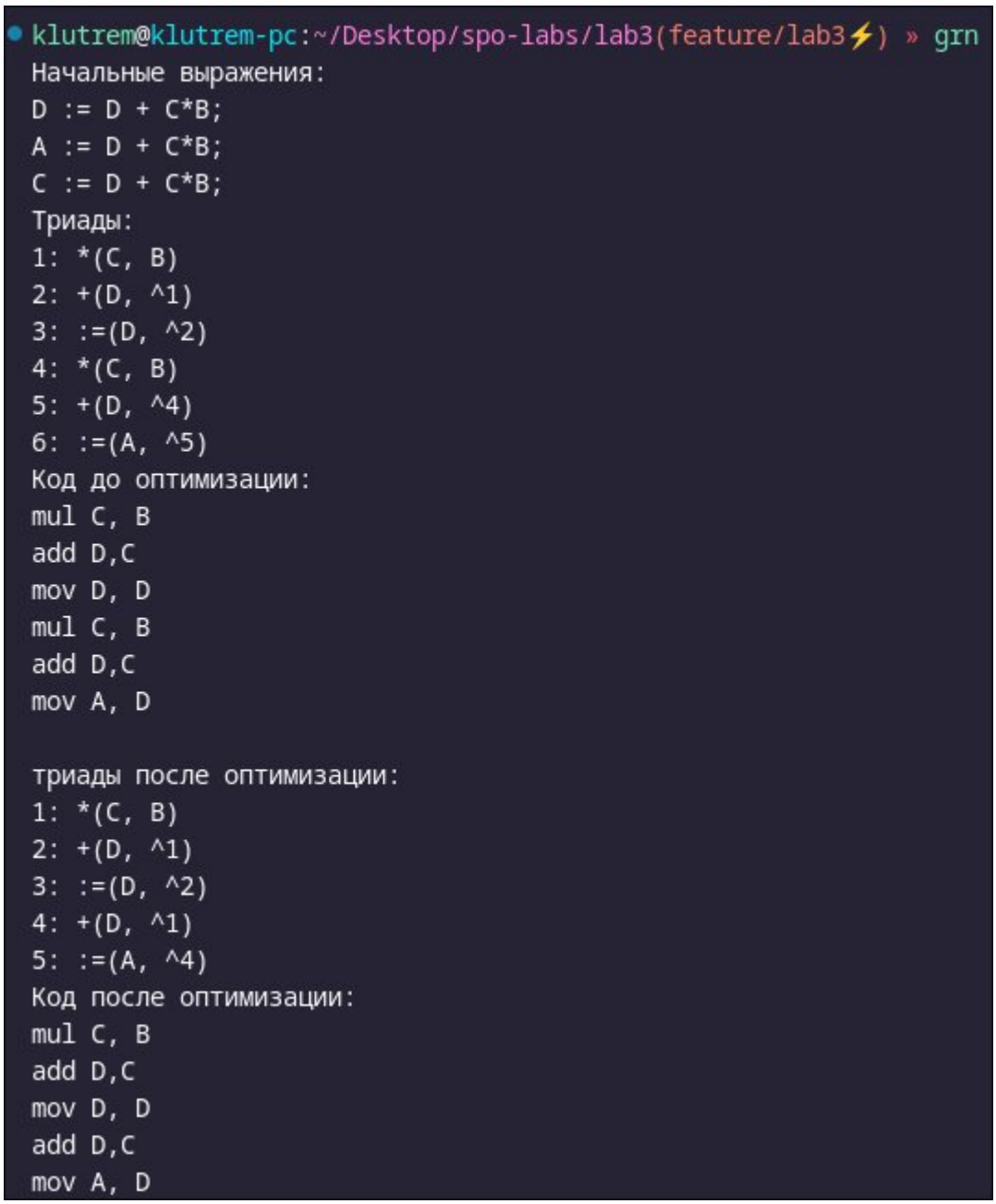

Рисунок 1 – результат выполнения программы 1

```
klutrem@klutrem-pc:~/Desktop/spo-labs/lab3(feature/lab3⚡) » grn
Начальные выражения:
a := 2;
a := 1 + 3;
b := 3;
c := 1;
a := (a + a) * a;
d := (b - c) / 2;
Триады:
1: :=(a, 2)
2: +(1, 3)
3: :=(a, ^2)
4: :=(b, 3)
5: :=(c, 1)
6: +(a, a)
7: *(^6, a)
8: :=(a, ^7)
9: -(b, c)
10: /(^9, 2)
11: :=(d, ^10)
Код до оптимизации:
mov a, 2
add 1,3
mov a, 1
mov b, 3
mov c, 1
add a,a
mul a, a
mov a, ^6
sub b,c
div b,2
mov d, ^9

триады после оптимизации:
1: :=(a, 2)
2: :=(a, 4)
3: :=(b, 3)
4: :=(c, 1)
5: :=(a, 32)
6: :=(d, 1)
Код после оптимизации:
mov a, 2
mov a, 4
mov b, 3
mov c, 1
mov a, 32
mov d, 1
```

Рисунок 2 – результат выполнения программы 2