Министерство науки и высшего образования Российской Федерации

Федеральное государственное автономное образовательное   
учреждение высшего образования

Национальный исследовательский Нижегородский государственный университет им. Н.И. Лобачевского

Институт информационных технологий, математики и механики

**Отчет по лабораторной работе**

**«Интерпретатор»**

**Выполнил**:

студент группы 3823Б1ПМ1-1

Золкин И.А.

**Проверил**:

преподаватель каф. ВВСП,

Волокитин В.Д.

Нижний Новгород

2025

**Содержание**

[Постановка задачи 3](#_Toc199373165)

[Метод решения 4](#_Toc199373166)

[Руководство пользователя 8](#_Toc199373167)

[Описание программной реализации 12](#_Toc199373168)

[Подтверждение корректности 18](#_Toc199373169)

[Результаты экспериментов 19](#_Toc199373170)

[Заключение 21](#_Toc199373171)

# Постановка задачи

Необходимо разработать интерпретатор, принимающий на вход код программы, соответствующий распознаваемому интерпретатором синтаксису. Входная программа должна выполнится построчно и выполнить описанные команды.

При создании допускается использование сторонних библиотек, не выполняющих поставленную задачу, в том числе библиотек, позволяющих описывать грамматику интерпретируемого языка.

Требования к языку программирования и синтаксису интерпретируемого языка не установлены.

# Метод решения

Для реализации исполнения инструкций создана следующая архитектура:

Создается и запускается один контролирующий объект, содержащий элементы хранения кода, сигналов условий, текущего выражения для возврата из функций, флаг дебага, несколько хэндлеров для хранения имен и положений отдельных исполняемых блоков.

Также контролирующий объект содержит стек вызовов, предназначенный для хранения вычисляемых объектов, полученных в ходе выполнения программы или считывания, и маркеров. Нулевой адрес стека считается невалидным и обозначается специальной меткой.

После нулевой метки стека вызовов сохраняются глобальные переменные. Глобальными переменными считаются все переменные, созданные до вызова функции int::main(::);

При исполнении true\_main (вызове int::main(::)) в контролирующем объекте переключается флаг и запоминается положение в стеке последней добавленной глобальной переменной.

Изначально интерпретатор должен считать текст входной программы и обработать команды включения других файлов в текст программы. Включение сторонних файлов осуществляется рекурсивно, вставкой кода подключаемого файла и возвращением указателя на начало обрабатываемого кода.

При загрузке кода интерпретируемой программы текст считывается построчно, если внутри строки обнаруживается знак завершения строки “;”, то строка делится по данному разделителю. В каждую строку вставляются дополнительные пробелы, разделяющие отдельные лексические единицы. Если во время вставки разделяющих пробелов указатель встречает двойные кавычки, то вставка не выполняется до встречи следующих, не экранированных двойных кавычек. К одинарным кавычкам такое правило не применяется. Обработанный код сохраняется в специальном контейнере.

В ходе выполнения программы строчки кода обрабатываются по одной, последовательно, с возможным перемещением указателя следующей строки на любую другую строку, в соответствии с командой.

После определения очередной выполняемой строчки происходит преобразование её в набор лексем. Каждая лексема имеет несколько основных показателей: текст лексемы, тип лексемы, возвращаемый лексемой тип, количество аргументов лексемы, сигнатура лексемы, сигнатура возвращаемого лексемой типа. Также каждая лексема имеет несколько мест хранения для разных типов данных и адреса ссылок.

Преобразование каждой строчки идёт независимо от остальных. Происходит разбитие строки по пробелам, если встречаются двойные кавычки, то пробелы игнорируются до встречи неэкранированных двойных кавычек. При встрече одинарных кавычек идет ожидание либо символа, либо двух символов в случае экранирования.

При встрече типа данных идет ожидание либо новой функции, либо новой переменной, причем тип сохраняется. Функция в отличии от переменной имеет круглые скобки после себя. При встрече нераспознанной лексемы делается предположение что это имя уже объявленной переменной и идет проверка на то, что переменная действительно была объявлена. Также нераспознанная лексема проверяется на возможность стать литералом.

При встрече треугольной открывающейся скобки ‘<’, идет проверка, есть ли перед ней операнды. В положительном случае знак становится оператором сравнения, иначе становится обрамляющей скобкой возвращаемого типа функции.

При вызове функции явно указывается возвращаемый тип, это нужно чтобы потом составить полную сигнатуру вызываемой функции. Следовательно, контроль за возвращаемым значением полностью перекладывается на автора интерпретируемой программы.

При встрече лексической единицы, начинающейся на двойные кавычки, выдвигается утверждение, что это строка. Сохранение строки идет цельной лексемой, ограниченной неэкранированными двойными кавычками.

Операторы cin/cout игнорируются из-за сложности создания экземпляра istream/ostream классов.

При встрече операторов, на данном этапе сигнатура составляется не полностью, например: “operator+”. По умолчанию у операторов два аргумента, но в случае унарного минуса и логического отрицания аргумент один.

Если после лексемы, распознанной как имя переменной стоит открывающаяся квадратная скобка, то лексема определяется как имя массива. Если переменная была помечена как новая, то массив соответственно будет тоже новым. Если квадратная скобка относится к новому массиву, то она тогда определяет выделение памяти, иначе является оператором обращения к массиву.

После разбития строки определяется необходимость в её выполнении. Всего есть три случая, которые обрабатываются итеративно конечными автоматами.

Если строка является частью объявления новой функции, то идет определение сигнатуры функции, сохранение имен аргументов в объявлении функции и сохранение адреса первой инструкции функции. Остальные строчки, относящиеся к функции, игнорируются.

Если строка относится к блоку if или while, то для каждой глубины вложенности создается счетчик правильной скобочной последовательности, а также для каждого уровня if/while хранится signal (результат сравнения), определяющий необходимость в выполнении полученной строки.

Когда распределяющий блок решает, что строку необходимо выполнить, строка проходит преобразование в обратную польскую запись (дальше RPN). Круглые и квадратные скобки обрабатываются вне очереди одинаковым образом – при встрече закрывающей скобки идет выталкивание лексем из стека до момента встречи соответствующей открывающей скобки. В случае квадратных скобок в результат добавляется открывающая квадратная скобка, которая сама по себе является оператором выделения или обращения к памяти. Приоритетность операций определена в отдельном отображении.

После преобразования в RPN строка передается исполнителю, в котором содержится определение всех базовых типов и операций в виде полных сигнатур.

Каждая RPN строка выполняется поэлементно.

Если обрабатываемая лексема является новой переменной, то производится проверка на ее наличие в текущей зоне видимости и глобальном секторе и в стеке создается блок новой переменной.

Если лексема является обращением к переменной, то производится поиск в своей зоне видимости и глобальном секторе. В положительном случае на верхушку стека кладется копия переменной или ссылочный блок.

Если лексема является новым массивом, то создается один блок массива, вне зависимости от его размера, если проверка на его отсутствие прошла успешно.

Если лексема является обращением к массиву, то производится поиск в текущей зоне видимости и глобальном секторе и в положительном случае на вершину стека кладется ссылочный блок.

Если лексема является литералом, то идет его распознавание и помещение на вершину стека.

Если лексема не относится к перечисленным видам, то она считается исполняемой. Происходит считывание количества аргументов лексемы arg\_c, и составление полной сигнатуры, анализируя последние arg\_c элементов на вершине стека. После составления полной сигнатуры производится поиск по предопределенным операциям, если лексема не является именем функции.

В случае не распознавания лексемы в качестве предопределенной операции происходит проверка по типу лексемы.

Если лексема является if или while, то в память управляющего объекта добавляется signal (результат сравнения).

Если лексема является оператором выделения памяти, то из стека вызовов считывается количество выделяемой памяти и создается n-1 объект, т.к один был создан при объявлении имени.

Если лексема является оператором очистки памяти delete, то производится постановка указателя на начало памяти удаляемого объекта и подсчет занимаемых ячеек стека, которые помечаются “FOR DELETE”. Дальше отдельным циклом ищем обьект “FOR DELETE” и удаляем его, останавливая после итерации.

Вызов функции не может осуществляться в одиночном режиме, т.е результат функции обязательно должен быть присвоен переменной. На момент исполнения функции известна её полная сигнатура: возвращаемыйтип::имяфункции::(::типпеременной1::типпеременной2::…::).

При вызове происходит перемещение указателя следующей строки на адрес первой строки функции, хранящегося в памяти управляющего объекта. В стек вызовов дублируется arg\_c элементов с вершины стека и переименуются в соответствии с объявлением функции, добавляется метка возвращения из функции, содержащая адрес возврата. В память управляющего элемента сохраняется невыполненная часть RPN данной строки.

При возвращении из функции параллельно идет проталкивание результата ниже по стеку и удаление блоков, созданных в стеке во время работы функции. При достижении метки возврата считывается адрес возврата и удаляются arg\_c блоков до метки и метка. Вызывается дополнительная итерация исполнения RPN, сохраненной в процессоре при вызове функции. После исполнения идет перемещение указателя следующей инструкции по считанному адресу.

Интерпретатор поддерживает режим дебага, для его активации нужно запускать интерпретатор с любым дополнительным аргументом командной строки.

В режиме дебага выводится обработанный код программы, с подсветкой зеленым цветом следующей выполняемой строки. При выполнении каждой строки интерпретатор ожидает нажатие любой клавиши на клавиатуре для продолжения работы. Нажатием клавиши ‘1’ можно вывести стек вызовов, с пронумерованными позициями и полными описаниям содержимого данных блоков.

# Руководство пользователя

Запуск интерпретатора производится командой “interpret.exe <filename>” для запуска в обычном режиме и командой “interpret.exe <filename> <something>” для запуска в режиме дебага, где filename – путь до файла с кодом интерпретируемой программы. В коде интерпретируемой программы запрещается использовать ключевые слова c/c++ и ключевые слова интерпретируемого языка в качестве имен переменных, функций, типов.

Общая инструкция по синтаксису интерпретируемого языка:

Переменные

Объявлять переменные можно с явным указанием значения и без. Сначала указывается тип переменной, потом имя переменной. Запрещается повторное использование одного имени для объявления переменных в рамках одной функции или одного глобального пространства, т.е. вне функций может быть одна переменная с “имя1” и при исполнении внутри функции может быть одна переменная c “имя1”. При этом каждый вызов функции рассматривается за новую область видимости, где может быть объявлена переменная с “имя1”.

Объявление переменной без инициализации выглядит как “<type> <name>;”, перечисление имен переменных через запятую, подобно синтаксису c++ нежелательно.

Объявление переменной с инициализацией выглядит как “<type> <name> = <varName/Literal/arrayElement>”:

double a = 4.6;

int a = b[4];

float g = r;

Для использования переменной, нужно писать её имя. Переменная ищется по имени в текущей области видимости (функции) и в глобальном сегменте.

При этом новой областью видимости не является блок if/while, при создании переменных в данных блоках нужно их вручную удалять командой “delete <varName>;”.

Типы данных

Доступны int, char, double, long\_long (с нижним подчёркиванием), uchar, bool, string, float. После исполнения операций возвращаемый тип соответствует левому операнду, если не описано иного. Таким образом можно делать приведение типов:

int perevod = 0;

double t = 4;

Выражение perevod + t возвратит значение типа int, выражение t + perevod возвратит значение типа double.

Операции присвоения между разными типами данных возможны, но при сужении типа, возможна потеря данных. Например, если char приравнять целое число int, которое больше типа char.

Нельзя ставить f после литерала для обозначения float, все литералы по умолчанию считаются либо long long либо double.

Нельзя делать последовательное присвоение a=b=c;

Управление памятью, массивы, ссылки

Есть возможность создавать многомерные массивы, размерность ограничена памятью и производительностью. Синтаксис следующий “<type> <arrayName>[arraySize];”, для создания многомерного массива достаточно приписать квадратные скобки с размерностью “<type> <arrayName>[arraySize1][arraySize2]…[arraySizeN];”. Объявление массива всегда происходит без инициализации, т. е. после закрывающейся квадратной скобки строка должна заканчиваться.

Обращение к элементу массива происходит ссылкой на относительную позицию в стеке. Используется индексация элементов, начинающаяся с 0. Пример выражений с обращением к массиву:

int a = g[32] + fe[0] – t[6][2];

de[3][6] = 1-4 + r[3];

При использовании имени массива без индексации, происходит работа с последним элементом в массиве. При использовании индексации к имени переменной происходит работа по ссылке, с точки зрения реализации производительность лучше не становится, поэтому лучше так к переменным не обращаться, так как делает код менее читабельным.

Вызов функций

При вызове функций обязательно указывается возвращаемый тип в треугольных скобках, интерпретатор ищет функцию по полному совпадению возвращаемого типа, имени функции и типам аргументов. Синтаксис вызова функции, треугольные скобки являются частью синтаксиса: “<returnType> funcName(arg1, arg2, …, arg n)”. При вызове функции необходимо куда-либо присваивать её значение. Пример:

double res = <double>rongekutta(x1, x2, x3, x4) + <double>eiler(y1,x2);

Для передачи массивов и переменных по ссылкам в объявлении функции используются квадратные скобки с указанием относительного адреса начала памяти:

int meow(int cats[0], int dogs[0])…

Причем передавать надо схожим образом:

R = <int>meow(cats[0], dogs[0]);

Каждая функция обязана иметь return <any>, даже пустая.

Подключение файлов и дополнительные функции

Есть возможность рекурсивно вставлять другие файлы с кодом, для этого внутри кода прописывается “# INSERT\_FILE <относительный адрес подставляемого файла>;”. Файлы будут вставляться пока не выполняться INSERT\_FILE из всех файлов.

Для повышения первичного функционала написаны дополнительные функции и переменные. В математическом модуле доступны cos, sin, решето Эратосфена, работа с векторами и двумерными матрицами, в алгоритмическом есть поразрядная сортировка типа int и перевод массива int размера n в массив uchar размером 4n.

Строки

Возможно использование строк как для хранения текста. Возможно присвоение string массиву char и наоборот, размер массива char постоянен.

char s[5];

s[0] = "eto pterodaktel\tpo imeni \'dinozavr\'";

string a = s[0];

теперь a = “eto p”.

Ввод/вывод

Вывод осуществляется “cout << outputData1 << outputData2 << .. << outputDataN << endl;”, можно выводить строки с экранированными символами, переменные, ключевой символ endl.

Ввод осуществляется “cin >> inpData1 >> inpData2 >>.. >> inpDataN >> endl;”

Циклы и ветвление

Для проверки условий доступна конструкция if-else:

if(условие)

{

//code1

}

else

{

//code2

}

Можно упускать блок else:

if(условие)

{

//code1

}

Для создания циклов доступно только while:

while(условие)

{

//code1

}

Блоки if и while можно вкладывать друг в друга. Нельзя создавать переменные внутри блоков, если очень хочется, то перед концом блока необходимо переменную удалить вручную командой delete.

# Описание программной реализации

Логически программа разделена на несколько компонентов, каждый из которых выполняет какую-либо функцию. Основной элемент управления описывается в файле ‘processor.h’.

class Processor {

public:

bool \_\_DEBUG\_\_STATE\_\_ = false;

std::vector<std::vector<Lexem>> Prev\_Expr;

size\_t GLOBAL\_SECTOR\_LAST;

bool GLOBAL\_SCAN;

std::vector<bool> if\_signals;

CodeSegment code\_segment;

CallStack call\_stack;

handleClassNames Class\_names\_list;

handleFuncNames Function\_names\_list;

Processor(const std::string& input\_file);

void run();

};

\_\_DEBUG\_\_STATE\_\_ является флагом, который определяет в каком режиме запущена программа – обычном или дебага.

Prev\_Expr хранит невыполненную часть RPN при вызове функции.

GLOBAL\_SECTOR\_LAST – хранит указатель на последнюю глобальную переменную.

if\_signals – хранит результаты проверки условий while и if.

code\_segment – место для хранения обработанного кода программы.

call\_stack – стек вызовов.

Метод run() запускает работу интерпретатора.

class\_names\_list – хранит информацию о типах данных.

Function\_names\_list – хранит информацию о существующих функциях, их адресах и аргументах.

При создании экземпляра данного класса происходит считывание файла и заполнение экземпляра класса CodeSegment:

class CodeSegment {

size\_t i;

std::vector<std::string> code\_rows;

public:

std::string true\_main = "<int>main();";

CodeSegment() = default;

CodeSegment(const std::vector<std::string>&);

std::string get\_next\_row();

std::pair<std::string,size\_t> get\_next\_row(size\_t ind);

void print\_all();

bool isEmpty();

size\_t get\_index();

void insert\_input\_point();

void set\_index(size\_t ind);

size\_t size();

};

i – указатель на текущую, еще не выполненную строчку кода.

code\_rows – массив строк кода интерпретируемой программы.

true\_main – строка, вставляется в конец программы, запускает функцию int::main::(::).

get\_next\_row(..) – возвращает строку на которую указывает i либо относительно ind и увеличивает i на единицу.

print\_all() – выводит весь код на экран с подсветкой текущей строки.

isEmpty() – проверяет, остались ли еще строки на выполнение.

get\_index() – возвращает текущее i.

insert\_input\_point() – ставит true\_main в конец кода программы.

set\_index() – устанавливает i.

size() – возвращает количество строк кода.

class handleFuncNames {

std::map<std::string, size\_t> Position;

std::map<std::string, std::vector<std::string>> Args;

public:

std::string last\_function\_signature;

void NewFunc(const std::string& signa, size\_t pos);

void DelFunc(const std::string& signa);

bool FuncExist(const std::string& signa);

size\_t GetPosFunc(const std::string& signa);

std::vector<std::string> get\_Args(const std::string& signa);

void set\_Args(const std::string& signa, const std::vector<std::string>& args);

};

Position – содержит позиции в коде объявленных функций.

Args – содержит имена аргументов функций.

last\_function\_signature – сигнатура последней обрабатываемой функции.

Остальные методы осуществляют добавление/удаление/управление Position и Args по сигнатуре функции.

class handleClassNames {

std::set<std::string> class\_names;

std::map<std::string, size\_t> class\_sizes;

public:

handleClassNames();

void add\_class\_name(const std::string& name);

bool isClass(const std::string& name);

void add\_class\_size(const std::string& name, size\_t sz);

size\_t get\_class\_size(const std::string& name);

};

class\_names – хранит имена типов.

class\_sizes – хранит размеры типов.

handleClassNames() – позволяет инициализировать размеры встроенных типов.

Остальные два метода позволяют добавлять и получать размеры типа.

class CallStack {

public:

std::vector<MemoryBlock> stack;

void push(const MemoryBlock& l);

MemoryBlock top();

void pop();

size\_t size() const;

void print\_all();

};

stack – место хранения элементов стека вызовов.

push – добавляет блок памяти на вершину стека.

top - возвращает блок памяти с вершины стека.

pop – удаляет блок памяти с вершины стека.

size – возвращает размер стека.

print\_all – выводит стек на экран в красном обрамлении, используется в режиме дебага.

class MemoryBlock {

public:

bool new\_var;

size\_t sz;

Lexem title;

size\_t array\_pointer;

std::vector<Lexem> data;

Lexem end;

MemoryBlock();

MemoryBlock(const Lexem& t, const Lexem& d, const Lexem& e, size\_t s\_);

MemoryBlock& operator=(const MemoryBlock& right);

size\_t size();

};

new\_var – определяет, является ли блок памяти оригиналом или копией.

sz – размер блока памяти.

title – определяет имя блока памяти, размер данных, адрес ссылки.

array\_pointer – вспомогательный указатель для работы с массивом.

data – основные данные блока памяти.

end – определяет конец блока памяти, может содержать дополнительную информацию.

size – возвращает sz.

В процессе исполнения экземпляр класса Processor создает экземпляр класса Lexem\_Queue\_Assembler, предназначенный для отправки инструкций на исполнение.

class Lexem\_Queue\_Assembler {

Processor\* processor;

long long arg\_brackets\_is\_working;

size\_t cnt\_brackets\_eq;

int state;

std::vector<std::string> args;

std::vector<std::string >class\_name\_space;

std::string signature;

std::vector<long long> states;

std::vector<long long> brackets;

std::vector<size\_t> while\_addr;

std::vector<std::vector<Lexem>> expressions;

public:

size\_t instruction\_definer();

bool continue\_mode;

std::vector<Lexem> lexem\_queue;

Lexem\_Queue\_Assembler(Processor\* p);

void push\_back(const Lexem& lexem);

};

arg\_brackets\_is\_working – массив для отслеживания правильных скобочных последовательностей в объявлении функций.

class\_name\_space – предназначено для обозначения класса в сигнатурах, в настоящей реализации не используется.

signature – вспомогательная строка для формирования сигнатур.

state отслеживает состояние автомата распознавание объявления функции.

states – отслеживает состояния автомата распознавания while/if.

brackets – отслеживает правильные скобочные последовательности в вложениях if/while.

while\_addr – отслеживает адреса возврата в блоках while.

expressions – удерживает выражения while.

continue\_mode – определяет, продолжать выполнять хранящуюся после возврата из функции инструкцию или приступить к следующей.

lexem\_queue – хранит очередь непереведенных в rpn лексем.

push\_back – добавляет в очередь лексему.

instruction\_definer() – переводит инструкции в rpn и отправляет их на выполнение.

size\_t Lexem\_Queue\_Assembler::instruction\_definer() {

auto rpn\_form = translate\_to\_rpn(lexem\_queue);

if (continue\_mode) {

rpn\_form = lexem\_queue;

}

Ariphmetic\_module calculator(rpn\_form, processor);

continue\_mode = false;

calculator.Calculate(lexem\_queue, \*this);

lexem\_queue.clear();

return 0;

}

class Ariphmetic\_module {

std::vector<std::string >class\_name\_space;

std::vector<Lexem> rpn\_form;

Processor\* processor;

public:

Ariphmetic\_module(const std::vector<Lexem>& rpn, Processor\* p);

void Calculate(std::vector<Lexem>& lexem\_queue, Lexem\_Queue\_Assembler& lexem\_queue\_asm);

size\_t int\_calc(Lexem& rpn\_);

size\_t int\_calc2(Lexem& rpn\_);

size\_t bool\_calc(Lexem& rpn\_);

size\_t char\_calc(Lexem& rpn\_);

size\_t float\_calc(Lexem& rpn\_);

size\_t double\_calc(Lexem& rpn\_);

size\_t string\_calc(Lexem& rpn\_);

size\_t long\_long\_calc(Lexem& rpn\_);

size\_t uchar\_calc(Lexem& rpn\_);

};

Ariphmetic\_module предназначен для последовательного выполнения инструкций из RPN.

Calculate выполняет все инструкции из rpn\_form.

Функции ниже предназначены для выполнения встроенных операций для заданных типов по полным сигнатурам.

# Подтверждение корректности

Для подтверждения корректности работы программ, использовались некоторые базовые алгоритмы перебора, вывода, ввода, вызова функций, алгоритма бинарного возведения в степень, решета Эратосфена, поразрядной сортировки, перемножения матриц.

Программа перемножения матриц (файл MATH.txt находится в .\include):

# INSERT\_FILE MATH.txt

int main(){

int a[2][2];

int b[2][2];

int c[2][2];

a[0][0] = 1;

a[0][1] = 2;

a[1][0] = 3;

a[1][1] = 4;

b[0][0] = 11;

b[0][1] = 22;

b[1][0] = 33;

b[1][1] = 44;

cout << a[0][0] << ' ' << a[0][1] << endl << a[1][0] << ' ' << a[1][1] << endl;

cout << b[0][0] << ' ' << b[0][1] << endl << b[1][0] << ' ' << b[1][1] << endl;

0 = <int>mult\_matrix(a[0][0], b[0][0], c[0][0], 2,2,2);

cout << c[0][0] << ' ' << c[0][1] << endl << c[1][0] << ' ' << c[1][1] << endl;

return 0;

}

Вывод программы:

1 2

3 4

11 22

33 44

77 110

165 242

# Результаты экспериментов

Эксперимент проводился на алгоритме решета Эратосфена для N чисел. Код алгоритма на интерпретируемом языке:

int eratosphen(int n, int prime[0])

{

int ind = 2;

while(ind < n+1){

prime[ind] = 1;

ind = ind + 1;

}

int j;

ind = 2;

while(ind<=n){

if(prime[ind]){

if(ind\*ind <= n){

j = ind\*ind;

while(j<=n){

prime[j] = 0;

j = j + ind;

}

}

}

ind = ind+1;

}

return 0;

}

Для замера времени использовалась библиотека chrono. Производился замер работы метода void Processor::run(). Полученные данные видны на графике зависимости времени от числа элементов.

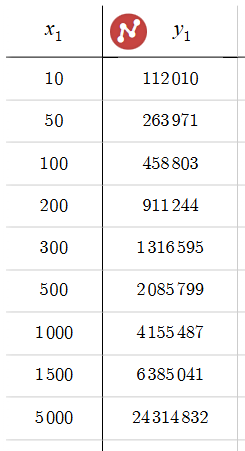


Рис. точки

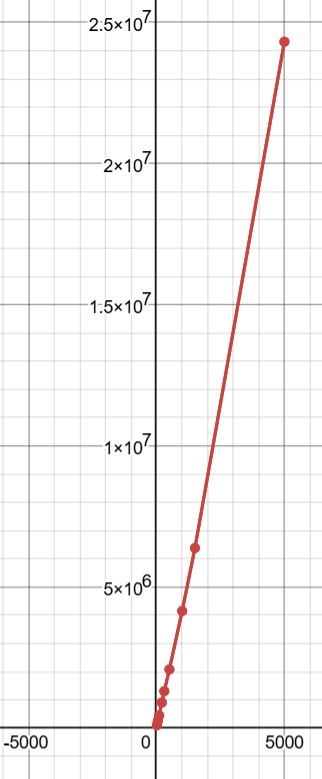


Рис. график

Результат эксперимента показал очень низкую производительность интерпретатора.

# Заключение

Был создан интерпретатор с функционалом, позволяющим реализовывать решения без использования ООП. Интерпретатор был проверен на корректность и на производительность. Производительность оказалась очень низкой, что ожидаемо – каждая операция сопровождается дополнительным копированием в стеке вызовов, что значительно замедляет выполнение инструкций.