МИНИСТЕРСТВО ОБРАЗОВАНИЯ И НАУКИ РОССИЙСКОЙ ФЕДЕРАЦИИ

федеральное государственное бюджетное образовательное учреждение

высшего образования
«Тольяттинский государственный университет»

Институт математики, физики и информационных технологий
(наименование института полностью)

Кафедра «Прикладная математика и информатика»
(наименование кафедры)

01.03.02 Прикладная математика и информатика
(код и наименование направления подготовки, специальности)

Системное программирование и компьютерные технологии
(направленность (профиль)/специализация

БАКАЛАВРСКАЯ РАБОТА

на тему Разработка интерпретатора и пиринговой вычислительной системы

Студент	И.С. Беляев (И.О. Фамилия)	(личная подпись)
Руководитель	А.В. Шляпкин (И.О. Фамилия)	(личная подпись)
Консультанты	М.А. Четаева (И.О. Фамилия)	(личная подпись)

Допустить к защите

Заведующий кафедрой к.т.н., доцент, А.В. Очеповский ________________
(ученая степень, звание, И.О. Фамилия) (личная подпись)

«_____»________________________20_____г.

Тольятти 2018

АННОТАЦИЯ

Название квалификационной работы: «Разработка интерпретатора и пиринговой вычислительной системы».

Объектом квалификационной работы является исследование процесса вычислений в пиринговых сетях.

Предметом исследования является автоматизация процесса пиринговых вычислений.

Целью работы является разработка интерпретатора и пиринговой вычислительной системы.

Работа состоит из введения, трёх глав и заключения.

Во введении определены актуальность темы, цели и задачи, поставленные в работе, объект и предмет исследования.

Первая глава посвящена: формированию требований к интерпретатору и системе пиринговых вычислений, анализу недостатков существующих пиринговых вычислительных систем и интерпретаторов.

Вторая глава посвящена проектированию интерпретатора и пиринговой вычислительной системы.

Третья глава посвящена разработке интерпретатора и системы пиринговых вычислений. В этой главе выбираем средства для быстрого развертывания системы пиринговых вычислений, описанию процессов интерпретатора и системы пиринговых вычислений, тестированию реализованного интерпретатора и пиринговой вычислительной системы.

В заключении подводятся основные итоги и выводы по работе.

Выпускная квалификационная работа состоит из 49 страниц. В работе: рисунков – 17, таблиц – 11, список литературы состоит из 21 источников. Работа состоит из введения, 3 глав, заключения, списка используемой литературы и приложения.

Результатом работы является интерпретатор и пиринговая вычислительная система.

ABSTRACT

The title of the bachelor’s thesis is Development of an interpreter and a peer-to-peer computing system.

The object of the study was the process of peer-to-peer computing.

The subject of the study was the automation of a peer-to-peer computing system and an interpreter.

The aim of the work was the development of an interpreter and a peer-to-peer computing system.

In the introduction, the urgency of the topic, the goals and tasks set in the work, the object and the subject of the study were determined.

The first part is devoted to the formation of requirements for an interpreter and a system of peer-to-peer computing, as well as to the analysis of shortcomings of peer-to-peer computing systems and interpreters.

The second part is about the design of an interpreter and a peer-to-peer computing system.

The third part describes the development of the interpreter and the system of peer-to-peer computing. This chapter is devoted to selecting the means for implementing the system of peer-to-peer computing, describing the processes of the interpreter and the peer-to-peer computing system, testing the implemented interpreter and the peer-to-peer computing system.

In conclusion, the main results and conclusions on the work were given.

The bachelor’s thesis consists of an explanatory note on 48 pages, introduction, three chapters, and a conclusion. The work includes 17 figures, 11 tables, a list of 21 references, including 19 foreign references, and 2 appendices.

The result of the work is the interpreter and the peer-to-peer computing system.

СОДЕРЖАНИЕ

ВВЕДЕНИЕ 6

1 АНАЛИЗ СИСТЕМ ПИРИНГОВЫХ ВЫЧИСЛЕНИЙ И ИНТЕРПРЕТАТОРОВ И ФОРМИРОВАНИЕ ТРЕБОВАНИЙ К РАЗРАБАТЫВАЕМОЙ СИСТЕМЕ И ИНТЕРПРЕТАТОРУ 9

1.1 Анализ систем пиринговых вычислений и формирование требований к разрабатываемой системе 9

1.2 Анализ интерпретаторов и формирование требований к разрабатываемому интерпретатору 16

2 ПРОЕКТИРОВАНИЕ ИНТЕРПРЕТАТОРА И ПИРИНГОВОЙ ВЫЧИСЛИТЕЛЬНОЙ СИСТЕМЫ 18

2.1 Функциональное моделирование системы пиринговых вычислений.... 18

2.2 Архитектура интерпретатора 20

3 РАЗРАБОТКА ИНТЕРПРЕТАТОРА И ПИРИНГОВОЙ ВЫЧИСЛИТЕЛЬНОЙ СИСТЕМЫ 29

3.1 Выбор средств для быстрого развертывания пиринговой вычислительной системы 29

3.2 Формальное описание процесса в пиринговой вычислительной системе 30

3.3 Формальное описание интерпретатора 32

3.4 Тестирование системы пиринговых вычислений 36

ЗАКЛЮЧЕНИЕ 40

СПИСОК ИСПОЛЬЗУЕМОЙ ЛИТЕРАТУРЫ .... **Ошибка! Закладка не определена.**

ПРИЛОЖЕНИЕ А. ФАЙЛ КОНФИГУРАЦИИ ДОКЕР КОНТЕЙНЕРА....... 43

ПРИЛОЖЕНИЕ Б. ПРОГРАМНЫЙ КОД ИНТЕРПРЕТАТОРА 44

ВВЕДЕНИЕ

Несмотря на широкое распространение крупных облачных платформ и ежегодно-снижающуюся стоимость на облачные вычисления, они всё ещё остаются централизованными с точки зрения управления. Это несёт за собой целый ряд вопросов: от приватности данных и защищенности централизованных серверов от взлома, до зависимости от корпорации. Поэтому сегодня актуальны разработки систем децентрализованных вычислений путём объединения свободных вычислительных мощностей. На сегодняшний день существует несколько решений, но у них имеются свои недостатки. В данной работе проведем анализ существующих решений, выявим их недостатки, на основе анализа разработаем пиринговую вычислительную систему.

В системе пиринговых вычислений важно обеспечить безопасность исполняемого программного кода. Многие языки программирования имеют встроенные методы отправки кроссдоменных запросов. Этот функционал можно использовать для проведения DDoS атак в разрабатываемой системе пиринговых вычислений. Поэтому необходимо ограничить возможности исходного языка. Для этого спроектируем и реализуем интерпретатор.

Актуальность данной работы заключается в том, что современные децентрализованные вычисления медленные и дорогие, по сравнению с централизованными, поэтому необходимо разработать пиринговую вычислительную систему.

Объектом квалификационной работы является исследование процесса вычислений в пиринговых сетях.

Предметом исследования является автоматизация процесса пиринговых вычислений.

Целью работы является разработка интерпретатора и пиринговой вычислительной системы.

Для достижения поставленной цели необходимо решить следующие задачи:

1. Сформировать требования для пиринговой вычислительной системы и интерпретатора.
2. Проанализировать существующие пиринговые вычислительные системы и интерпретаторы и выявить их недостатки.
3. Смоделировать собственную пиринговую вычислительную систему.
4. Разработать архитектуру интерпретатора.
5. Выбрать средства для реализации пиринговой вычислительной системы.
6. Разработать интерпретатор и пиринговую вычислительную систему.
7. Протестировать пиринговую вычислительную систему.

Выпускная квалификационная работа состоит из 49 страниц. В работе: рисунков – 17, таблиц – 11, список литературы состоит из 21 источников. Работа состоит из введения, 3 глав, заключения, списка используемой литературы и приложения.

Первая глава посвящена: формированию требований к интерпретатору и системе пиринговых вычислений, анализу недостатков существующих пиринговых вычислительных систем и интерпретаторов.

Вторая глава посвящена проектированию интерпретатора и пиринговой вычислительной системы.

Третья глава посвящена разработке интерпретатора и системы пиринговых вычислений. В этой главе выбираем средства для быстрого развертывания системы пиринговых вычислений, описанию процессов интерпретатора и системы пиринговых вычислений, тестированию реализованного интерпретатора и пиринговой вычислительной системы.

В заключении подводятся основные итоги и выводы по работе.

Результатом работы является интерпретатор и пиринговая вычислительная система.

В списке используемой литературы перечисляются учебные пособия и литература на иностранном языке.

1 АНАЛИЗ СИСТЕМ ПИРИНГОВЫХ ВЫЧИСЛЕНИЙ И ИНТЕРПРЕТАТОРОВ И ФОРМИРОВАНИЕ ТРЕБОВАНИЙ К РАЗРАБАТЫВАЕМОЙ СИСТЕМЕ И ИНТЕРПРЕТАТОРУ

1.1 Анализ систем пиринговых вычислений и формирование требований к разрабатываемой системе

На сегодняшний день существует много систем пиринговых вычислений рассмотрим некоторые из них:

Climate Prediction. Проект, который использует вычислительные мощности пользователей для предсказания погоды на 50 лет в перед. Основывая свою предсказания на схемах и теориях. Цели проекта:

- выявление эффективности и погрешности используемых сегодня методик;
- при нахождении эффективных методик использовать их для прогноза погода на большой срок.

RainbowCrack. Проект формирует радужные таблицы для всех популярных алгоритмов хэширования. Радужные таблицы хранят все возможные хэши и их значения. Если иметь хэш, то при помощи быстрого бинарного поиска можно получить значение этого хэша.

Все рассмотренные системы в основном узкоспециализированные и не позволяют пользователю загрузить собственный программный код.

Сегодня существует децентрализованная система Ethereum. Ethereum (ETH) – это распределенная вычислительная платформа с открытым исходным кодом, основанная на блочных связях, и операционная система, в которой реализованы функции смарт-контракта.

Смарт-контракте – это алгоритм, предназначенный для цифровой поддержки, проверки или обеспечения выполнения переговоров или исполнения контракта. Смарт-контракты позволяют выполнять надежные транзакции без третьих сторон. Эти транзакции являются прослеживаемыми и необратимыми.

За проведение вычислений в ETH расплачиваются газом. Чем больше вычисления – тем больше газа необходимо заплатить. К примеру, на Python можно сложить два числа 1 миллион раз за 0,05 секунды. В ETH за это придется заплатить 0.09 ETH что в переводе на сегодняшний курс $53.68. За подобные вычисления это огромные деньги. Поэтому не рационально пользоваться этой системой для емких вычислений.

Проведем эксперимент, сравним скорость вычислений в системе Ethereum и локально, на языке JavaScript.

Для сравнения используем методом наименьших квадратов и построим график аппроксимации. В таблице 1 находятся данные по Ethereum системе.

Таблица 1 – Результаты вычислений системы Ethereum

X	0.1	1	1.5	2.5	5	7.5	10
Y	54.07	58.3	66.47	70,16	128.75	210.12	333.04

Методом наименьших квадратов находим квадратичную функцию, которая наилучшим образом приближает эмпирические данные основанные на реализованной системе.

Уравнение регрессии:

$$y = ax^2 + bx + c \quad (1.1)$$

Коэффициенты a, b, c, d найдём как решение системы:

$$\begin{aligned} a \quad x_i^2 + b \quad x_i + nc &= \quad y_i\,, \\ a \quad x_i^3 + b \quad x_i^2 + c \quad x_i &= \quad x_i y_i\,, \\ a \quad x_i^4 + b \quad x_i^3 + c \quad x_i^2 &= \quad x_i^2 y_i\,, \end{aligned} \quad (1.2)$$

В целях более компактной записи переменную «счетчик» можно опустить, поскольку и так понятно, что суммирование осуществляется от 1 до n = 7. В таблице 2 представлен расчет нужных сумм.

Таблица 2 – Расчет необходимых сумм системы Ethereum

x_i	y_i	x_i^2	x_i^3	x_i^4	$x_i y_i$	$x_i^2 y_i$
0,1	54,07	0,01	0,001	0,0001	5,407	0,5407
1,0	58,30	1,00	1,000	1,0000	58,300	58,3000
1,5	66,47	2,25	3,375	5,0625	99,705	149,5575
2,5	70,16	6,25	15,625	39,0625	175,400	438,5000
5,0	128,75	25,00	125,000	625,0000	643,750	3218,7500
7,5	210,12	56,25	421,875	3164,0625	1575,900	11819,2500
10,0	333,04	100,00	1000,000	10000,0000	3330,400	33304,0000
$\sum x_i$	$\sum y_i$	$\sum x_i^2$	$\sum x_i^3$	$\sum x_i^4$	$\sum x_i y_i$	$\sum x_i^2 y_i$
27,6	920,91	190,76	1566,876	13834,1876	5888,862	48988,8982

Таким образом, получили следующую систему:

$$190{,}76a + 27{,}6b + 7c = 920{,}91$$
$$1566{,}876a + 190{,}76b + 27{,}6c = 5888.862$$
$$13834{,}1876a + 1566{,}876b + 190{,}76c = 48988.898$$

Далее решаем систему методом Крамера:

$$\Delta = \begin{matrix} 190.76 & 27.6 & 7 \\ 1566.876 & 190.76 & 27.6 \\ 13834.1876 & 1566.876 & 190.76 \end{matrix} = -306525{,}2432$$

$$\Delta_a = \begin{matrix} 920.91 & 27.6 & 7 \\ 5888.867 & 190.76 & 27.6 \\ 48988.898 & 1566.876 & 190.76 \end{matrix} = -827101{,}012408$$

$$\Delta_b = \begin{matrix} 190.76 & 920.91 & 7 \\ 1566.876 & 5888.867 & 27.6 \\ 13834.1876 & 48988.898 & 190.76 \end{matrix} = -222331{,}1112792$$

$$\Delta_c = \begin{matrix} 190.76 & 27.6 & 920.91 \\ 1566.876 & 190.76 & 5888.867 \\ 13834.1876 & 1566.876 & 48988.898 \end{matrix} = -16909719{,}131008$$

$$a = \frac{\Delta_a}{\Delta} = \frac{-827101{,}012408}{-306525{,}2432} = 2{,}69831288207587335175793443 42887 \approx 2{,}698$$

$$b = \frac{\Delta_b}{\Delta} = \frac{-222331{,}1112792}{-306525{,}2432} = 0{,}7253272486081498686826586294 0191 \approx 0{,}725$$

$$c = \frac{\Delta_c}{\Delta} = \frac{-16909719{,}131008}{-306525{,}2432} = 55{,}165828936231637577573578449085 \approx 55{,}166$$

Выполняем проверку. Подставим найденное решение в систему уравнений.

$$920{,}91 = 920{,}91$$
$$5888.862 = 5888.862$$
$$48988.898 = 48988.898$$

Искомая аппроксимирующая функция: $y = 2{,}698x^2 + 0{,}725x + 55{,}166$ – из всех кубических функций экспериментальные данные наилучшим образом приближает именно она.

Посчитаем среднюю ошибку аппроксимации по формуле:

$$A = \frac{1}{n} \quad \frac{y_i - y_i}{y_i} * 100\% \qquad (1.3)$$

Полученные результаты внесем в таблицу 3.

Таблица 3 – Расчет средней ошибки аппроксимации

x_i	0,1	1	1,5	2	5	7,5	10
y_i	54,07	58,30	66,47	70,16	128,75	210,12	333,04
y_i	55.265	58.589	62.324	73.841	126.241	212.366	332.216
$\frac{y_i - y_i}{y_i}$	0,0221	0,0050	0,0624	0,0525	0,0195	0,0107	0,0025

Средняя ошибка аппроксимации равна 2,49%.

Теперь найдем кубическую функцию, которая наилучшим образом приближает эмпирические данные основанные на JavaScript алгоритме. В таблице 4 данные по JavaScript алгоритму.

Таблица 4 – Результаты вычислений по JavaScript алгоритму

X	0.1	1	1,5	2,5	5	7,5	10
Y	0,351	2,73	5,625	15,073	58,537	125,835	226,019

В таблице 5 представлен расчет нужных сумм.

Таблица 5 – Расчет необходимых сумм по JavaScript алгоритму

x_i	y_i	x_i^2	x_i^3	x_i^4	$x_i y_i$	$x_i^2 y_i$
0,1	0,35	0,01	0,001	0,0001	0,035	0,0035
1,0	2,73	1,00	1,000	1,0000	2,730	2,7300
1,5	5,63	2,25	3,375	5,0625	8,438	12,6563
2,5	15,07	6,25	15,625	39,0625	37,683	94,2063
5,0	58,54	25,00	125,000	625,0000	292,685	1463,4250
7,5	125,84	56,25	421,875	3164,0625	943,763	7078,2188
10,0	226,02	100,00	1000,000	10000,0000	2260,190	22601,9000
x_i	y_i	x_i^2	x_i^3	x_i^4	$x_i y_i$	$x_i^2 y_i$
27,6	434,17	190,76	1566,876	13834,1876	3545,523	31253,1398

Таким образом, получили следующую систему:

$$190{,}76a + 27{,}6b + 7c = 434{,}17$$
$$1566{,}876a + 190{,}76b + 27{,}6c = 3545{,}5226$$
$$13834{,}1876a + 1566{,}876b + 190{,}76c = 31253{,}1398$$

Далее решаем систему методом Крамера:

$$\Delta = \begin{matrix} 190.76 & 27.6 & 7 \\ 1566.876 & 190.76 & 27.6 \\ 13834.1876 & 1566.876 & 190.76 \end{matrix} = -306525{,}2432$$

$$\Delta_a = \begin{matrix} 434.17 & 27.6 & 7 \\ 3545.5226 & 190.76 & 27.6 \\ 31253.1398 & 1566.876 & 190.76 \end{matrix} = -681725{,}5623624$$

$$\Delta_b = \begin{matrix} 190.76 & 434.17 & 7 \\ 1566.876 & 3545.5226 & 27.6 \\ 13834.1876 & 31253.1398 & 190.76 \end{matrix} = -80964{,}18809576$$

$$\Delta_c = \begin{matrix} 190.76 & 27.6 & 434.17 \\ 1566.876 & 190.76 & 3545.5226 \\ 13834.1876 & 1566.876 & 31253.1398 \end{matrix} = -114783{,}5674928$$

$$a = \frac{\Delta_a}{\Delta} = \frac{-681725{,}5623624}{-306559{,}462136} = 2{,}2240437859063739256826075327945 \approx 2{,}224$$

$$b = \frac{\Delta_b}{\Delta} = \frac{-80964{,}18809576}{-306559{,}462136} = 0{,}26410598300124100006357404160422 \approx 0{,}2641$$

$$c = \frac{\Delta_c}{\Delta} = \frac{-114783{,}5674928}{-306559{,}462136} = 0{,}37442513335921166857719502741527 \approx 0.3744$$

Выполняем проверку. Подставим найденное решение в систему уравнений.

$$434.17 = 434.17$$
$$3545{,}5226 = 3545{,}5226$$
$$31253{,}1398 = 31253{,}1398$$

Таким образом, искомая аппроксимирующая функция: $y = 2.224x^2 + 0.2641x + 0.3744$ – из всех кубических функций экспериментальные данные наилучшим образом приближает именно она.

Посчитаем среднюю ошибку аппроксимации. Полученные результаты внесем в таблицу 6.

Таблица 6 – Расчет средней ошибки аппроксимации

x_i	0,1	1	1,5	2	5	7,5	10
y_i	0,351	2,73	5,625	15,073	58,537	125,835	226,019
y_i	0,423	2,8625	5,775	14,93	57,2949	127,455	225,415
$\frac{y_i - y_i}{y_i}$	0,2051	0,0485	0,0267	0,0095	0,0212	0,0129	0,0027

Средняя ошибка аппроксимации равна 4,67%.

По получившимся функциям аппроксимации рассчитаем результаты алгоритмов в системе Ethereum и скрипта JavaScript на большие нагрузки. В таблице 7 представлены результаты.

Таблица 7 – Прогноз результатов системы Ethereum и скрипта JavaScript

x	f_1	f_2
50	6836.416	5573.5794
100	27107.666	22266.7844
150	60868.916	50079.9894
200	108120.166	89013.1944

На рисунке 1 изображен график, оранжевым цветом обозначена функция скрипта JavaScript, синим цветом обозначена функция Ethereum системы, ось x – количество простых чисел (1:10000), ось y – количество затраченных секунд.

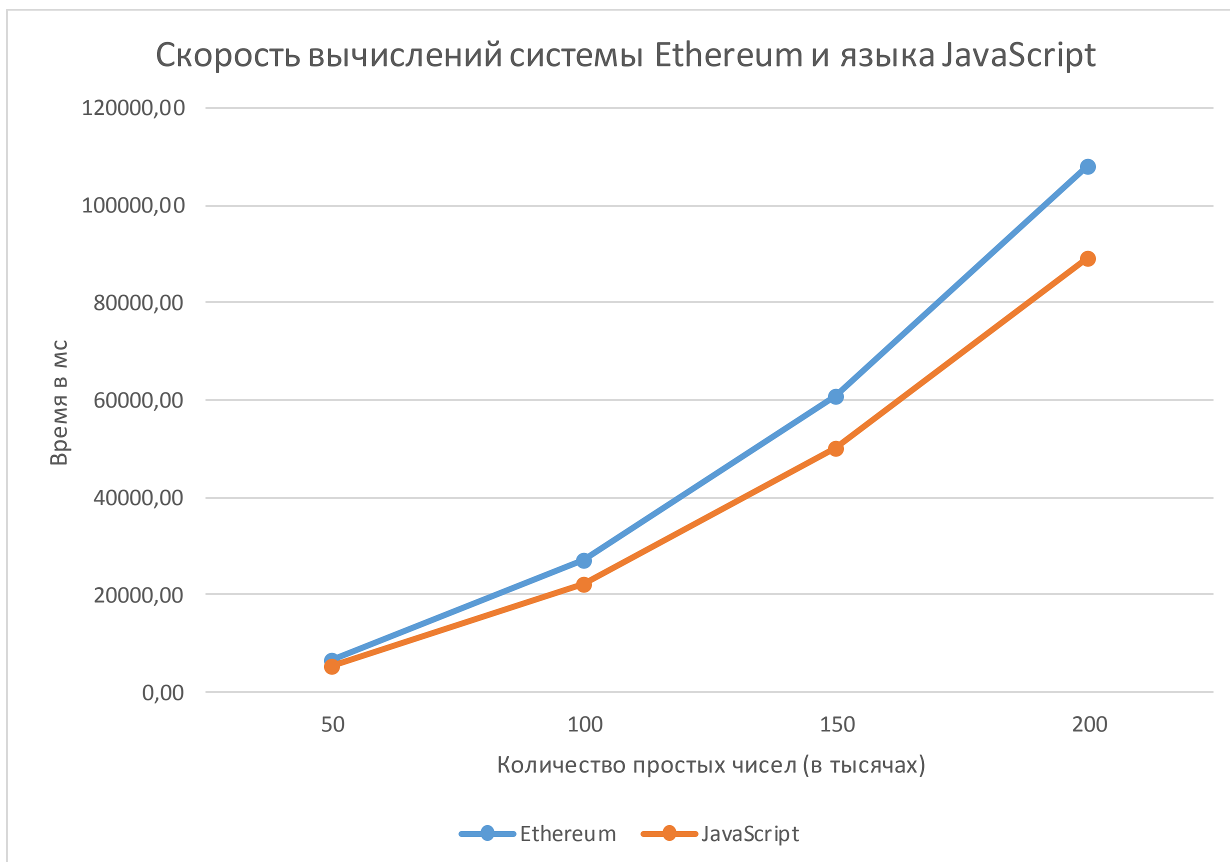

Рисунок 1 – График сравнения скорости выполнения алгоритма в системе Ethereum и скрипта JavaScript

В итоге, у Ethereum имеются свои недостатки:

- за вычисления в Ethereum сети необходимо расплачиваться криптовалютой. Чем больше платишь за вычисления, тем быстрее их произведут. Отсюда вытекает дороговизна вычислений;
- медленные вычисления по сравнению с динамическим языком программирования JavaScript.

Исходя из недостатков Ethereum системы, сформируем требования для разрабатываемой пиринговой вычислительной системы. Система должна:

- принимать на вход программный код, выполнять его и на выходе отдавать полученный результат;
- производить быстрые и бесплатные вычисления;
- безопасно исполнять программный код обособлено от файловой системы.

1.2 Анализ интерпретаторов и формирование требований к разрабатываемому интерпретатору

Для исполнения программного кода необходим интерпретатор JavaScript языка. В языке программирования JavaScript есть встроенный интерфейс «XMLHttpRequest». Этот интерфейс отвечает за отправки http запросов. Также, в нем есть возможность отправлять кроссдоменные http запросы. За счет этого у пользователей есть возможность устраивать DDoS атаки на различные сервисы. Из этого не доброжелательные пользователи могут воспользоваться другими участниками системы для организации DDoS атак на различные сервисы.

Для проведения математических расчетов, интерпретатор должен иметь следующий функционал:

- арифметические операции;
- логические операции;
- функции;
- условный оператор;
- цикл с предусловием.

На основе требований к интерпретатору проведем анализ существующих интерпретаторов.

Исходя из поставленных требований проанализируем существующие интерпретаторы языка JavaScript.

NeilFraser JS-Interpreter. Плюсы рассматриваемого интерпретатора: реализованы все арифметические операции, логические операции, условные операторы, функции и циклы.

Минусы рассматриваемого интерпретатора: есть возможность отправлять кроссдоменные http запросы.

js.js. Плюсы рассматриваемого интерпретатора:

- интерпретатор с открытым исходным кодом;
- реализованы арифметические операции, логические операторы, условные операторы и функции;
- нет возможности отправлять http запросы.

Минусы рассматриваемого интерпретатора: не реализованы циклы, что не позволяет создавать алгоритмы.

Исходя из анализа, ни пиринговые вычислительные системы, ни интерпретаторы не отвечают поставленным требованиями. Поэтому было решено реализовать новую систему пиринговых вычислений и интерпретатор.

2 ПРОЕКТИРОВАНИЕ ИНТЕРПРЕТАТОРА И ПИРИНГОВОЙ ВЫЧИСЛИТЕЛЬНОЙ СИСТЕМЫ

2.1 Функциональное моделирование системы пиринговых вычислений

Рассмотрим архитектуру пиринговой вычислительной системы. Каждый пир (нода) непрерывно посылает запросы остальным нодам, до тех пор, пока не будет получен программный код, который необходимо выполнить. После получения программного кода, нода начинает выполнять его. По завершению выполнения программного кода результат отправляется ноде, чей программный код выполнился.

В системе существует две роли:

- нода с исходным программным кодом – нода, чей программный код необходимо выполнить;
- исполнитель программного кода – нода, которая выполняет программный код и отправляет результат ноде с исходным программным кодом.

Варианты использования:

- получение программного кода – исполнитель запрашивает программный код;
- исполнение программного кода – исполнение программного кода, полученного от клиента;
- отправка результата исполнения программного кода – отправка результата исполненного программного кода клиенту.

Построим диаграмму последовательности для отображения временных особенностей передачи и приему сообщений объектами. Основными элементами диаграммы последовательности являются объекты (прямоугольники с названиями объектов), вертикальные «линии жизни», которые отображают течение времени, прямоугольники, отражающие деятельность объекта или исполнения им определенной функции (прямоугольники на пунктирной «линии жизни»), и стрелки, которые

показывают обмен сообщениями между объектами. На рисунке 2 изображена диаграмма последовательности.

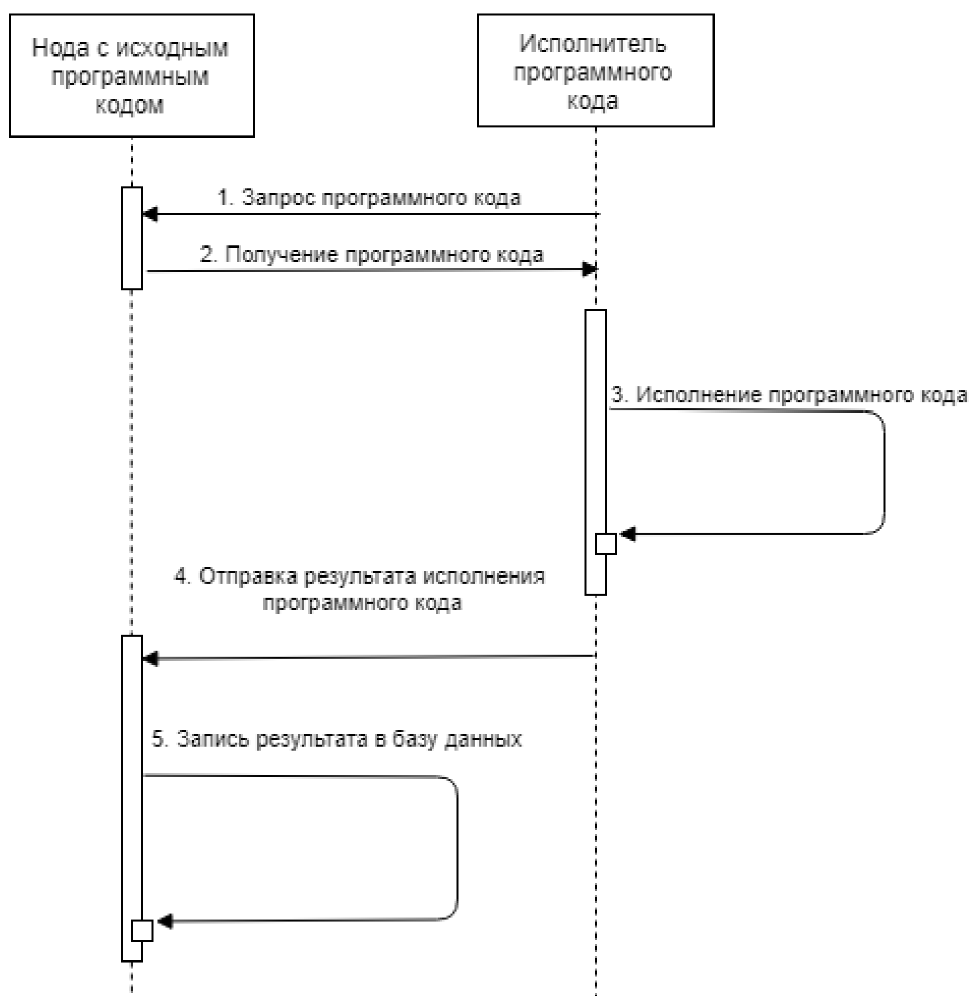

Рисунок 2 – Диаграмма последовательности пиринговой вычислительной системы

На данной диаграмме изображены следующие объекты: пир с исходным программным кодом, исполнитель программного кода, вертикальные «линии жизни», которые отображают течение времени, прямоугольники, отражающие деятельность объекта и стрелки, которые показывают обмен сообщениями между объектами.

Пиры между собой обмениваются сообщениями по http 2 протоколу.

Http 2 имеет несколько методов передачи данных. Чаще всего используются get и post запросы.

Get запросы являются идемпотентными по спецификации, то есть ожидается, что сервер должен отдавать одни и те же ответы на идентичные

запросы. Это позволяет кэшировать ответы. Параметры в данном методе передаются в URL запросе.

Post запрос в отличии от get не является идемпотентным, так что данный метод может возвращать разные ответы [17]. Так же, по данному методу могут передаваться файлы. Параметры данного запроса передаются в теле запроса.

Схема get и post запроса изображена на рисунке 3.

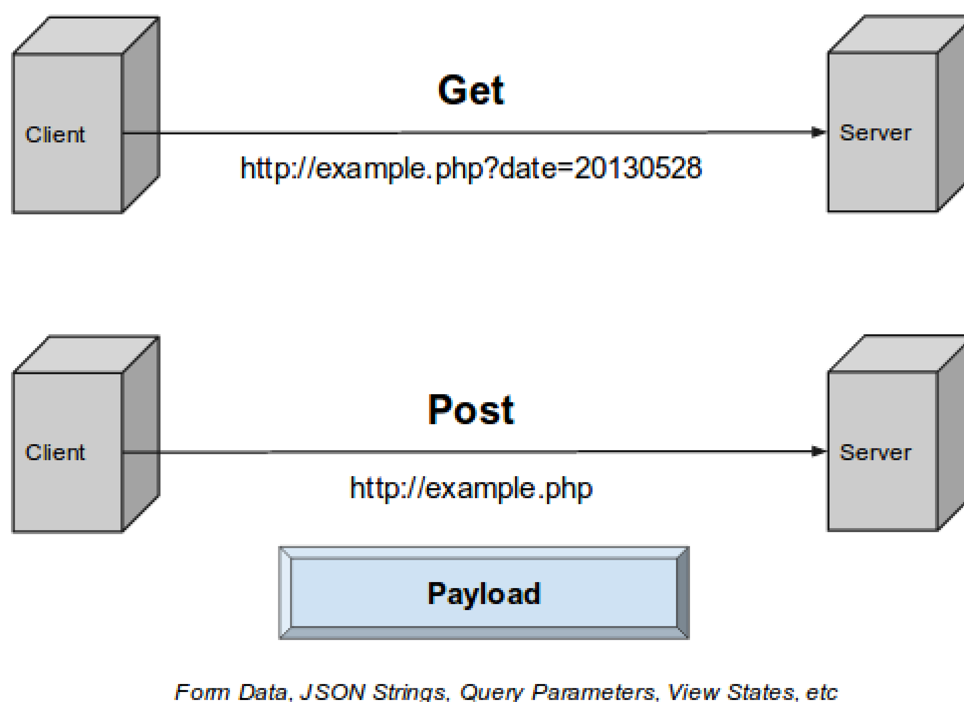

Рисунок 3 – Схема get и post запросов

В системе пиринговых вычислений используются get и post методы.

2.2 Архитектура интерпретатора

Для работы интерпретатора необходимо разбить выражения на лексемы. После нужно разбить лексемы на лексические классы (непересекающиеся подмножества). Таким образом, все идентификаторы могут рассматриваться как одинаковые при анализе. Размеры лексических классов различаются. Как правило, во многих языках программирования имеются следующие лексические классы:

- строковые литералы;
- идентификаторы;

- числовые константы.

Всем подмножествам присваиваются некоторое число, которое называют токеном (идентификатор лексического класса) или лексическим классом.

Некоторые языковые особенности препятствуют лексическому анализу. Например, в языках Фортран и Кобол требуется размещение конструкций языка на фиксированных позициях входной строки. Для корректности программы это играет важную роль. Если в Коболе в шестой колонке не поставить специальный символ, то следующая строка будет неправильно проанализирована. Почти во всех современных языках программирования используется свободное размещение программного кода. Многие языки программирования заимствуют правила из других языков программирования, к примеру использования символов. В Алгол 68 и Фортран пробелы значатся исключительно в строковых литералах.

Лексический анализатор должен иметь дополнительную информацию о текущей лексеме, так как чаще всего, несколько лексем принадлежат одному лексическому классу.

Таблица используется для хранения всех идентификаторов и констант. Затем происходит свертывание. Свертывание – процесс замены идентификатора на ссылку в эту таблицу.

Простейшая форма организации таблицы – массив указателей на строки. Но при этом подходе тормозят два основных процесса таблицы представлений: добавление нового элемента и поиск идентификатора в таблице. Самая частая задача в процессе компиляции - это поиск идентификатора в таблице, так как идет поиск для всех использующих вхождения идентификатора.

Следовательно, нужно максимально оптимизировать процесс для этой операции. Еще существует набор хэшированных списков, который является еще одной формой организации таблицы.

Для данной задачи берется некоторая функция расстановки (хэш-функция), выдающая некоторое число по данному идентификатору от 0 до N-1, где N – константа, называемая длинной оглавления. Затем идентификаторы с одинаковым хэш значением попадают в список. В результате для нахождения идентификатора, необходимо найти его хэш-значение в таблице. В итоге, для проверки на повторное появление идентификатора в программе, нужно найти его хэш-значение в таблице [2].

Новые идентификаторы стоит добавлять в начало хэш-списка, поскольку на практике новые идентификаторы используются не далеко от места его определения. Это повышает быстродействие алгоритма, так же упрощает реализацию стандартных правил видимости.

Большинство распознаваемых лексем имеют четко заданную структуру, за счет этого появляется возможность использовать теорию языков для решения задачи лексического анализа, для того, чтобы с помощью формализма описать цепочки, принимаемых на вход и по этому описанию автоматически генерировать лексический анализатор.

Определим операции над множествами:

$$PQ = \{x \mid x = yz, x \in P, y \in Q\} \quad (2.1)$$

$$P^k = \begin{cases} \varepsilon \ , \text{если } k = 0 \\ P^{k-1}P, \text{если } k > 0 \end{cases} \quad (2.2)$$

$$P^* = \bigcup_{i=0}^{\infty} P^i \quad (2.3)$$

Регулярные выражения соответствуют естественному классу распознавателей в виде конченых автоматов, так как праволинейные грамматики эквивалентны регулярным выражениям [6].

Существует ряд средств для создания лексических анализаторов. Почти все они основываются на регулярных выражениях. Традиционное средство для создания лексического анализатора – Lex, он состоит из Lex компилятора и языка.

Lex анализирует входной файл, который был подготовлен в спецификации лексического анализатора Lex, в результате анализатор отдает программу.

Lex-программы состоят из трех частей: описаний, правил трансляции и процедур. Каждая часть отделяется от следующей строкой, содержащей два символа %%.

Существует раздел описаний, который берет на себя описание переменных, констант, регулярных выражений. Раздел описаний содержит определение макросимволов (метасимволов), в виде: ИМЯ ВЫРАЖЕНИЕ.

Если регулярное выражение находит в тесте ИМЯ, то оно заменяется на ВЫРАЖЕНИЕ. Если строка описаний заключена в скобки или начинается с пробелов, то она копируется в выходной файл.

Регулярные определения – это последовательность определений вида:

$d_1 r_1$

...

$d_n r_n$

где каждое d_i – некоторое имя, а каждое r_i – регулярное выражение над алфавитом

$$d_1, \dots, d_n \ .$$

Правила трансляции – это операторы вида:

$p_1\{action_1\}$

...

$p_n\{action_n\}$

где p_i – регулярное выражение, $action_i$ – фрагмент программы, описывающий, какие действия должен выполнять лексический анализатор для лексемы, определяемой p_i.

Разбор продолжается до тех пор, пока не будет возвращено нулевое значение.

Далее рассмотрим входной язык Lex'а, а именно запись регулярных выражений. Регулярное выражение из одного символа представляет символ из входного алфавита. В кавычки можно брать символы, цепочки.

Допустимы три способа кодирования символа «a»: a, “a” и \a. После префикса \ записываются специальные символы [16].

Существует возможность определения класса символов:

- [A-Za-z] – любая буква;
- [0-9] или [0123456789] – любая цифра;
- [^0-7] – любая литера, кроме цифр от 0 до 7;
- . – любая литера, кроме \n.

В порядке убывания приоритета задается грамматика для записи регулярных выражений:

- <p>* – повторение 0 или более раз;
- <p>+ – повторение 1 или более раз;
- <p>? – необязательный фрагмент;
- <p><p> – конкатенация;
- <p>{m,n} – повторение от m до n раз;
- <p>{m} – повторение m раз;
- <p>{m,} – повторение m или более раз;
- ^<p> – фрагмент в начале строки;
- <p>$ – фрагмент в конце строки;
- <p>|<p> – любое из выражений;
- <p>/<p> – первое выражение, если за ним следует второе;
- (p) – скобки, используются для группировки.

После прохождения лексического анализатора полученные команды-метки проходят через парсер.

Существует 2 типа парсеров: на внедряемых в код без абстрактной схемы данных и на использующих формальное описание языка.

Преимущества описания грамматики:

- упрощается поддержка языка;

- упрощает обозревание структуры языка, т.к. в одному месте находятся все синтаксические конструкции. Без описания грамматики, велика вероятность запутаться в сотнях, а то и тысячах строках кода. Также, это позволяет легко определить эволюцию конструкций;
- просто отслеживать ошибки разных категорий – ошибки составления кода и синтаксические ошибки входных данных. Без отслеживания ошибок в коде могут содержаться противоречия, логические ошибки, мертвые участки и при этом компилироваться. При работе с данной схемой все ошибки проявляются на этапе генерации;
- дополнительная абстракция позволяет тестировать и отслеживать более высокоуровневые сущности, которые являются более конкретными и менее многочисленными чем простой поток символов. Так же абстракция упрощает исходный код [13].

У программирования без абстрактной схемы тоже есть плюс. Входной порог значительно ниже, чем с абстрактной схемой.

Анализатор может состоять из трех частей, как представлено на рисунке 4, или может быть цельным модулем, без разделения на компоненты, здесь нет лучшего решения.

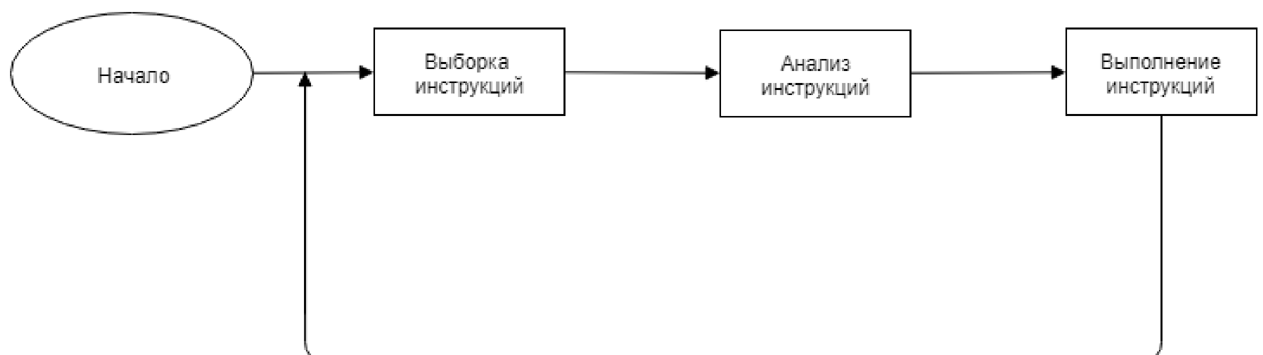

Рисунок 4 – Условная схема анализатора

В первую очередь нужно разобраться с составляющими. На вход лексического анализатора приходит поток символов заданного алфавита. Далее анализатор разбивает поток на терминалы или токены. Затем на основе правил составления символов второго иерархического порядка

(нетерминалов) и токенов синтаксических анализатор собирает дерево вывода. И в заключении происходит выполнение бизнес-логики и семантический анализ.

Лексический анализатор может быть интегрирован в синтаксическую часть, но как правило это не делают по ряду причин:

- сужается алфавит, что за собой влечет упрощение семантики;
- убирается один или несколько нижних уровней дерева при полном сохранении его свойств;
- собственно, лексический анализатор не должен брать на себя роль синтаксического, он не должен на «2*2» возвращать 4, но простейшие действие, к примеру формирование чисел, должен.

Рассмотрим рисунок 5, на котором изображен разбор дерева вывода выражения 12+34 в двух случаях:

DIGIT = '0'..'9'

NUMBER = NUMBER? DIGIT

ADD = NUMBER '+' NUMBER

SIGN = ('+' | '-') NUMBER

EXPR = ADD | SIGN

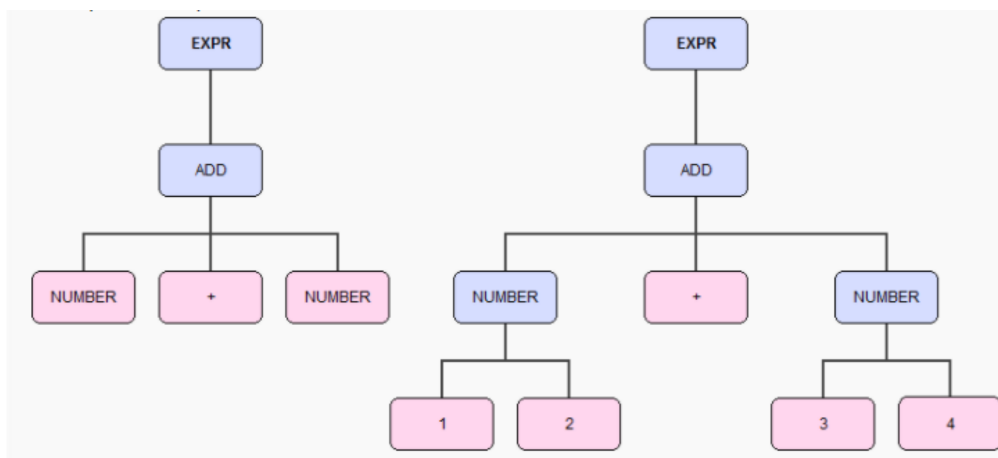

Рисунок 5 – Два дерева вывода выражения 12+34

На данном примере заметно, что для удобства анализ лучше разбивать на два этапа. Лексический анализатор позволяет использовать более эффективные алгоритмы по сравнению с разбором грамматик. Их главное

различие, помимо вышеуказанного эмпирического, это отсутствие правил вывода, вернее их можно представить неявно, но в начале будут стоять только знаки алфавита, и в правилах нет связи с остальными нетерминалами лексера. Это означает, что они самодостаточны и описывают исключительно ожидаемый поток символов [3].

Далее рассмотрим процесс интеграции семантической логики напрямую в синтаксический анализатор, минуя стадию построения дерева. Суть такова – намечаем точки, которые имеют семантическое значение и пишем код.

Для простой грамматики это хороший метод написания кода, но сложную грамматику так не описать.

Существует нисходящие (LL) и восходящие (LR) анализаторы.

Свои названия они берут из метода построение дерева вывода. Первый строит дереву сверху вниз, а второй снизу-вверх. Рассмотрим примеры этих алгоритмов.

Условно LR-парсер имеет стек, в котором хранятся последние поступившие терминалы и нетерминалы (символы), на каждом шаге, парсер пытается подобрать правило во время считывания очередного токена и подбирает правило, которое можно применить к набору символов с вершины стека, если находит, то он сворачивает один нетерминал в последовательность символов.

LL-парсер пытается сделать наоборот — для каждого входящего символа угадать к какому правилу он относится. Самый примитив — это выбор из альтернатив (например, EXPR может развернуться в ADD или в SIGN, то есть 2 альтернативы) по первому символу, и рекурсивный спуск с новым набором правил, которые продуцируют нетерминалы из выбранного пути. И так до тех пор, пока не правила не разложатся до терминалов [1].

Какой синтаксический анализатор использовать — дело вкуса. Практически по всем характеристикам они одинаковы.

После создания абстрактного синтаксического дерева исполнитель кода рекурсивно выполняет его команды-метки.

В данной главе произвели функциональное моделирование системы пиринговых вычислений. Рассмотрели архитектуру разрабатываемого интерпретатора.

3 РАЗРАБОТКА ИНТЕРПРЕТАТОРА И ПИРИНГОВОЙ ВЫЧИСЛИТЕЛЬНОЙ СИСТЕМЫ

3.1 Выбор средств для быстрого развертывания пиринговой вычислительной системы

Для удаленного исполнения программного кода важен вопрос безопасности. Необходимо учитывать, что исполняемый код может быть (ресурсоемким и есть необходимость выделить часть мощностей для системы пиринговых вычислений) вредоносным и целесообразно отгородить интерпретатор от файловой системы. Это можно сделать посредством виртуальных машин. Существует множество виртуальных машин как платных, так и бесплатных. Но у них у всех есть существенные минусы, они достаточно неудобно масштабируются, довольно много времени занимает процесс настройки виртуальной машины и установки операционной системы.

Сегодня существует универсальное решение – Docker. Docker – программное обеспечение для автоматизации развертывания и уравнения приложением в среде виртуализации на уровне операционной системы. Другими словами, он позволяет быстро развернуть виртуальную машину с приложением со всеми зависимостями.

Плюсы Docker:

- легковесность. Docker контейнеры имеют минимально необходимый набор софта для корректной работы всех контейнеров;
- кроссплатформенность. Docker контейнеры могут быть перенесены на любую операционную систему, на которой установлена Docker-служба;
- быстрота разработки. Достаточно просто указать необходимые модули и при их наличии Docker найдет их в Docker Hub и скачает, и установит зависимости;

- простота настройки. При запуске контейнера есть возможность задать определенное количество системных ресурсов для работы контейнера.

Поэтому используем Docker для быстрого развертывания системы пиринговых вычислений и интерпретатора.

3.2 Формальное описание процесса в пиринговой вычислительной системе

Для того чтобы занести в базу данных код, который необходимо выполнить, необходимо отправить post запрос с параметром code. Функция обработки запроса изображена на рисунке 6.

```
app.post('/add_code', function (req, res) {
    mysql.query(
        `INSERT INTO code_table (code) VALUES ("${req.body.code}")`,
        function(err, results) {
            if (err){
                res.send(err)
            } else {
                res.send(results);
            }
        }
    );
});
```

Рисунок 6 – Запрос добавления программного кода в базу данных

Для того, чтобы получить код у пира, который необходимо выполнить, необходимо отправить get запрос пиру, в теле ответа придет программный код. Функция обработчик изображена на рисунке 7.

```
app.get('/get_code', function (req, res) {
    mysql.query(
        `SELECT code FROM code_table WHERE id="${req.query.id}"`,
        function(err, results) {
            if (err) res.send(err);

            res.send(results);
        }
    );
});
```

Рисунок 7 – Запрос получение программного кода

После выполнения программного кода, нода оправляет результат выполнения ноде, от которой получила программный код. Запрос на отправку результата ноде изображен на рисунке 8.

```
app.post('/send_result', function (req, res) {
    mysql.query(
        `SELECT code FROM code_table WHERE id="${req.query.id}"`,
        function(err, results) {
            if (err) {
                res.send(err);
            }
            if (results === []) {
                res.send("Code doesn't exist");
            } else {
                mysql.query(
                    `UPDATE code_table SET result="${req.body.result}" WHERE
id="${req.body.id}"`,
                    function(err, results) {
                        if (err) {
                            res.send(err);
                        }
                        res.send(results);
                    }
                );
            }
        }
    );
});
```

Рисунок 8 – Отправка результата ноде

Для получения результата исполнения программного кода, необходимо отправить get запрос ноде, у которой храниться программный код и результат его исполнения. В запросе передается уникальный идентификатор программного кода (id). В теле ответа приходит результат исполнения программного кода. На рисунке 9 изображена функция обработчик запроса.

```
app.get('/get_result', function (req, res) {
    mysql.query(
        `SELECT result FROM code_table WHERE id="${req.query.id}"`,
        function(err, results) {
            if (err) res.send(err);
            res.send(results);
        }
    );
});
```

Рисунок 9 – Запрос получения результата исполнения программного кода

Выполнение программного кода происходит в Docker контейнере, в приложении А представлен файл конфигурации. В Docker контейнере содержится разработанный интерпретатор.

3.3 Формальное описание интерпретатора

Для написания интерпретатора понадобилось реализовать лексический анализатор, функцию для разбора команд (парсер) и исполнитель итогового кода.

Лексический анализатор принимает текст с командами языка AEL и возвращает список команд-меток (токены). Основа функции лексического анализатора изображена на рисунке 10.

```
const lex = function (input) {
    const tokens = [];
    // основная работа по лексическому анализу производится здесь
    return tokens;
};
```

Рисунок 10 – Основа функции лексического анализатора

Есть несколько разных типов команд-меток (токены):

- операторы: +-*/^%=();
- числа, записываемые цифрами: 0123456789;
- пробелы, которые лексический анализатор будет игнорировать: ' ';
- идентификатор переменных и функция, записываемых ка строки, состоящие из любых символов, кроме операторов, цифр и пробелов.

Разбитие на токены происходит с помощью регулярных выражений. На рисунке 11 изображены функции определения типа текущего символа.

```
let isOperator = function(c) {
    return /[+\-*\/\^%=(),]/.test(c);
};

let isDigit = function(c) {
    return /[0-9]/.test(c);
};

let isWhiteSpace = function(c) {
    return /\s/.test(c);
};

let isIdentifier = function(c) {
```

```
    return typeof c === "string" && !isOperator(c) && !isDigit(c) && 
!isWhiteSpace(c);
};
```

Рисунок 11 – Функции определения типа символа

Реализован простой цикл для сканирования текста с командами языка AEL. Понадобится переменная с именем “c”, которая будет хранить в себе текущий строковый символ. Также создана функция с названием advance для перехода вперед на один строковый символ в тексте с командами AEL и возвращения следующего строкового символа, и функция с названием addToken, которая добавляет команду-метку в список команд-меток.

Если символ в переменной “c” являться пробелом, то игнорируем его и алгоритм будет разбирать текст дальше. Если символ в переменной c будет являться оператором, то алгоритм добавит команду-метку оператора в список команд-меток и будет разбирать текст дальше. На рисунке 4 изображена проверка текущего символа на пробел.

Для упрощения работы в алгоритме определяется символ числа, как последовательность цифр, которые также могут быть разделены точкой для записи десятичных чисел.

Все остальные символы алгоритм считает идентификаторами переменных и функций. В случае возникновения непредвиденной ошибки будет выбрасываться исключение. После того, как алгоритм просканировал текст командами AEL, он добавляет завершающую команду-метку (токен) - "end", обозначающую конец.

Каркас функции парсера принимает на вход набор токенов, созданных лексическим анализатором и возвращает разобранное дерево. На рисунке 12 изображена функция парсера.

```
const parse = function (tokens) {
    const parseTree = [];

    // основная работа по разбору синтаксического дерева производится здесь

    return parseTree;
};
```

Рисунок 12 – Функция парсера

Было необходимо некое подобие таблицы символов для ассоциации символа с его силой присваивания: nud или led. Также понадобилась функция для ассоциации команды-метки с соответствующим символом.

Понадобился способ увидеть, что представляет текущая команда-метка и способ перейти к следующей команде-метке. Для этого реализована функция получения текущего токена и переход к следующему токену. Программный код этих функций изображен на рисунке 13.

```
let i = 0;
let token = function() {

    return interpretToken(tokens[i]);
};

let advance = function() {
    i++;
    return token();
};
```

Рисунок 13 – Функции получения текущего токена и переход к следующему токену

Далее нужно было реализовать функцию разбирающую синтаксическое дерево по разобранному выше принципу. Для определения операторов использовались инфиксные и префиксные функции.

После этого алгоритм определяет все арифметические операторы декларативным способом. Обращаю внимание, что оператор возведения в степень (^) имеет правую силу связывания (присвоении), которая меньше, чем левая сила связывания. Это потому что возведение в степень справа ассоциативно. Остались еще некоторые операторы, которые существуют только в виде разделения или конечной демаркации. Они не являются префиксами или инфиксами, так что добавляем их в таблицу символов.

Скобки и числа являются nud функцией, просто возвращающей свое содержимое.

Идентификаторы переменных и функций и оператор присваивания являются более сложными nud функциями, потому что имеют зависящее от контекста записи поведение.

Как цикл находит токен (end) он прекращает выполнения и переходит к функции evaluate. Окончание функции parse изображен на рисунке 14.

```
let parseTree = [];

while (token().type !== "(end)") {

    parseTree.push(expression(0));

}
return parseTree;
```

Рисунок 14 – Окончание функции parse

Исполнитель кода разбирает дерево, разобранное парсером и выполняет каждый его элемент, в результате чего выводит результат. Каркас функции исполнителя кода evaluate изображен на рисунке 15.

```
const evaluate = function (parseTree) {
    const parseNode = function (node) {
        // основная работа по исполнению кода производится здесь
    };
    const output = "";

    for (let i = 0; i < parseTree.length; i++) {
        const value = parseNode(parseTree[i]);

        if (typeof value !== "undefined") {output += value + "\n";}
    }

    return output;
};
```

Рисунок 15 – Основа функции evaluate

Далее определяем поведение всех операторов и предварительно определенных переменных и функций.

На рисунке 16 изображена функция calculate, которая принимает на вход выражение и выдает результат.

```
let calculate = function(input) {
    let tokens = lex(input);

    let parseTree = parse(tokens);

    return evaluate(parseTree);
```

```
};
```

Рисунок 16 – Функция calculate

Программный код интерпретатора представлен в приложении Б.

3.4 Тестирование системы пиринговых вычислений

Сравним получившуюся систему пиринговых вычислений с платформой Ethereum. Сравним скорость вычислений.

В пиринговой сети будет находиться 5 компьютеров. Их характеристики: Intel Pentium 4405u, 8gb ram ddr3; Intel Pentium g4560, 8 gb ram ddr4; Intel core 2 duo e8400, 5gb ram ddr2; Intel i3-6006u, 4 gb ram ddr3; Intel core i5-4690K, 8gb ram ddr3.

Для сравнения используем методом наименьших квадратов и по полученной функции аппроксимации построим график.

Тестирования проведем на алгоритме поиска простых чисел. За x возьмем количество простых чисел (в 10 тысячах), которые необходимо рассчитать. За y время, которое потратила система на вычисление. В таблице 8 занесены данные по разработанной системе.

Таблица 8 – Результаты вычислений системы пиринговых вычислений

X	0.1	1	1,5	2,5	5	7,5	10
Y	0.383	3.02	6.69	17.47	67.53	145.32	259.423

В таблице 9 представлен расчет нужных сумм.

Таблица 9 – Расчет необходимых сумм разработанной системы

x_i	y_i	x_i^2	x_i^3	x_i^4	$x_i y_i$	$x_i^2 y_i$
0,1	0,38	0,01	0,001	0,0001	0,038	0,0038
1,0	3,02	1,00	1,000	1,0000	3,020	3,0200
1,5	6,69	2,25	3,375	5,0625	10,035	15,0525
2,5	17,47	6,25	15,625	39,0625	43,675	109,1875
5,0	67,53	25,00	125,000	625,0000	337,650	1688,2500

7,5	145,32	56,25	421,875	3164,0625	1089,900	8174,2500
10,0	259,42	100,00	1000,000	10000,0000	2594,230	25942,3000

Предложение таблицы 9

x_i	y_i	x_i^2	x_i^3	x_i^4	$x_i y_i$	$x_i^2 y_i$
27,6	499,84	190,76	1566,876	13834,1876	4078,548	35932,0638

Таким образом, получили следующую систему:

$$\begin{gathered}\mathbf{190{,}76}a + 27{,}6b + 7c = \mathbf{499{,}836} \\ \mathbf{1566{,}876}a + \mathbf{190{,}76}b + 27{,}6c = \mathbf{4078{,}5483} \\ \mathbf{13834.1876}a + \mathbf{1566.876}b + \mathbf{190.76}c = \mathbf{35932{,}06383}\end{gathered}$$

Далее решаем систему методом Крамера:

$$\Delta = \begin{vmatrix} 190.76 & 27.6 & 7 \\ 1566.876 & 190.76 & 27.6 \\ 13834.1876 & 1566.876 & 190.76 \end{vmatrix} = -306525{,}2432$$

$$\Delta_a = \begin{vmatrix} 499.836 & 27.6 & 7 \\ 4078.5483 & 190.76 & 27.6 \\ 35932.06383 & 1566.876 & 190.76 \end{vmatrix} = -775672{,}8723$$

$$\Delta_b = \begin{vmatrix} 190.76 & 526.5 & 7 \\ 1566.876 & 4301.746 & 27.6 \\ 13834.1876 & 37916.7366 & 190.76 \end{vmatrix} = -172260{,}56721$$

$$\Delta_c = \begin{vmatrix} 190.76 & 27.6 & 526.5 \\ 1566.876 & 190.76 & 4301.746 \\ 13834.1876 & 1566.876 & 37916.7366 \end{vmatrix} = -70086{,}0978816$$

$$a = \frac{\Delta_a}{\Delta} = \frac{-775672{,}8723}{-306525{,}2432} = 2{,}530535052194357087781930516057 4 \approx 2{,}5305$$

$$b = \frac{\Delta_b}{\Delta} = \frac{-172260{,}56721}{-306525{,}2432} = 0{,}5619784048180472987550670998048 5 \approx 0{,}562$$

$$c = \frac{\Delta_c}{\Delta} = \frac{-70086{,}0978816}{-306525{,}2432} = 0{,}2286470672037623555827261146105 8 \approx 0{,}2286$$

Выполняем проверку. Подставим найденное решение в систему уравнений.

$$542.7 = 542.7$$
$$4431.954 = 4431.954$$
$$39066.9944 = 39066.9944$$

Таким образом, искомая аппроксимирующая функция: $y = 2{,}5305x^2 + 0.562x + 0.2286$ – из всех кубических функций экспериментальные данные наилучшим образом приближает именно она.

Для сравнения возьмем уже посчитанные данные Ethereum сети.

По получившимся функциям аппроксимации рассчитаем результаты систем на большие нагрузки. В таблице 10 представлены результаты систем на большие нагрузки. f_1 – реализованная пиринговая вычислительная система, f_2 – система Ethereum.

Таблица 10 – Результаты систем на большие нагрузки

x	f_1	f_2
50	6354.5786	6836.416
100	25361.4286	27107.666
150	57020.7786	60868.916
200	101332.6286	108120.166

Посчитаем среднюю ошибку аппроксимации. Полученные результаты внесем в таблицу 11.

Таблица 11 – Расчет средней ошибки аппроксимации

x_i	0,1	1	1,5	2	5	7,5	10
y_i	0,38	3,02	6,69	17,47	67,53	145,320	259,42
y_i	0,31	3,32	6,765	17,449	66,3	146,784	258,8986
$\frac{y_i - y_i}{y_i}$	0,1842	0,0993	0,0112	0,0012	0,0182	0,0101	0,002

Средняя ошибка аппроксимации равна 4,66%.

На рисунке 17 построен график, оранжевым цветом обозначена функция реализованной системы, синим цветом обозначена функция

Ethereum системы, ось x – количество простых чисел (в тысячах), ось y – количество затраченных миллисекунд.

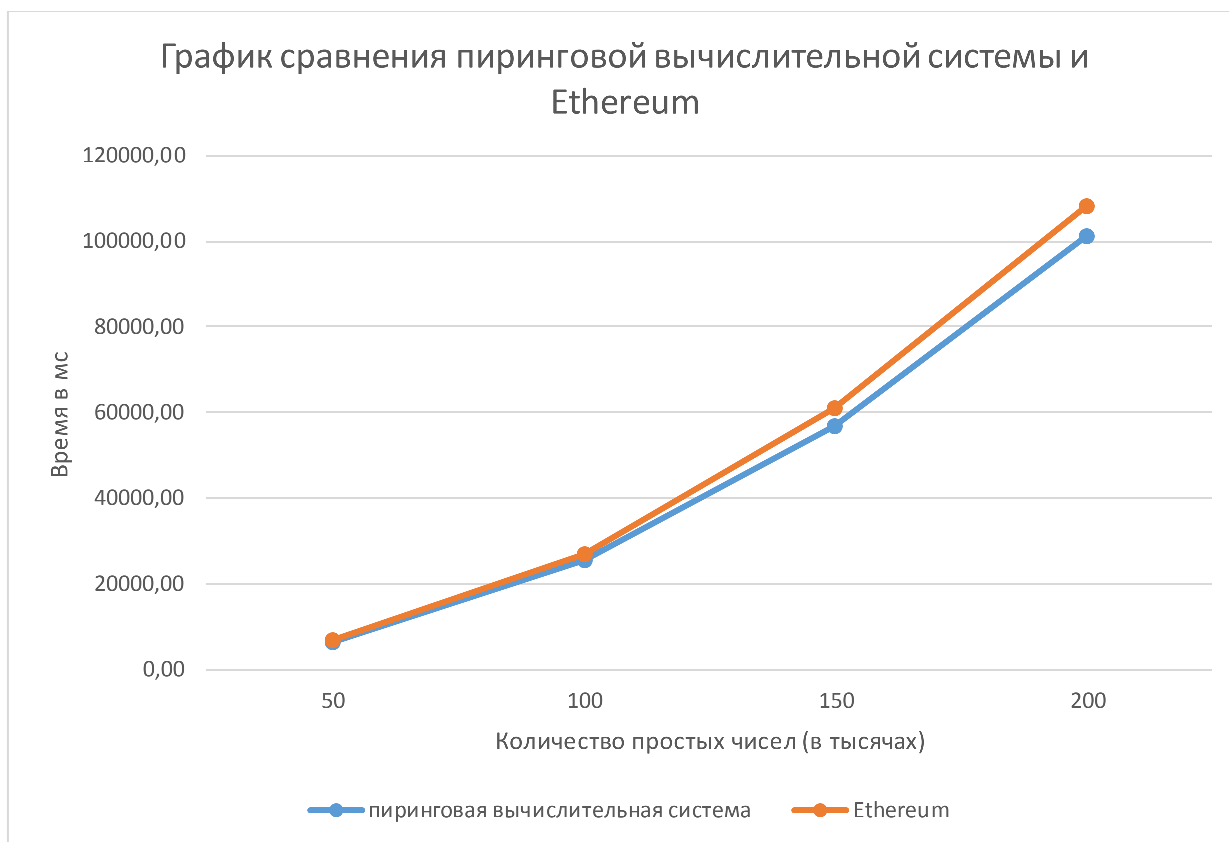

Рисунок 17 – График сравнения разработанной системы и системы Ethereum

По полученному графику, можно сделать вывод, что разработанная система пиринговых вычислений быстрее Ethereum системы.

ЗАКЛЮЧЕНИЕ

Во время выполнения выпускной квалификационной работы были сформированы требования к системе пиринговых вычислений, был проведен анализ существующих аналогов, были проанализированы недостатки существующих систем пиринговых вычислений. На основе недостатков было принято решение реализовать свою систему пиринговых вычислений и интерпретатор.

Была смоделирована пиринговая вычислительная система и разработана архитектура интерпретатора. Был выбран Docker, как средство для обособления системы пиринговых вычислений от файловой системы компьютерах. Реализована система пиринговых вычислений на фреймворке Express, платформы Node.js. Использовалась база данных MySQL. Интерпретатор реализован на языке JavaScript. Было проведено тестирование разработанного пиринговой вычислительной системы с системой Ethereum методом наименьших квадратов.

Итогом выпускной квалификационной работы является система пиринговых вычислений с собственным интерпретатором, позволяющая безопасно проводить трудоемкие вычисления на удаленных компьютерах в пиринговой сети.

Список используемой литературы

Учебники и учебные пособия

1. Компиляторы. Принципы, технологии и инструментарий / Ахо, Лам, Сети, Ульман, - 2 изд. Вильямс, 2016. - 1184 с.

2. Никлаус В. Построение компиляторов. - Москва: ДМК Пресс, 2016. - 192 с.

3. Aarne R. Implementing Programming Languages. College Publications, 2012. - 133 c.

4. Andy Mulholland, Jon Pyke, Peter Fingar Enterprise Cloud Computing: A Strategy Guide for Business and Technology Leaders. - 1 изд. Meghan-Kiffer Press, 2010. - 260 c.

5. Andy Oram Peer-to-Peer: Harnessing the Power of Disruptive Technologies. - 1 изд. - Sebastopol: O'Reilly Media, 2001. - 450 c.

6. Appel A. W. Modern Compiler Implementation in C. Princeton University, 2004. - 556 c.

7. Azat Mardan Practical Node.js: Building Real-World Scalable Web Apps. - 1 изд. - New York City: Apress, 2014. - 300 c.

8. John Buford, Heather Yu, Eng Keong Lua P2P Networking and Applications (Morgan Kaufmann Series in Networking (Hardcover)). - 1 изд. - Burlington: Morgan Kaufmann, 2008. - 408 c.

9. John Rhoton Cloud Computing Explained: Implementation Handbook for Enterprises. - 2 изд. Recursive Press, 2009. - 472 c.

10. Mario Casciaro, Luciano Mammino Node.js Design Patterns - Second Edition: Master best practices to build modular and scalable server-side web applications. - 2 изд. Packt Publishing - ebooks Account, 2016. - 526 c.

11. Michael J. Kavis Architecting the Cloud: Design Decisions for Cloud Computing Service Models (SaaS, PaaS, and IaaS). - 1 изд. - Hoboken: Wiley, 2014. - 224 c.

12. Muchnick S.S. Advanced Compiler Design and Implementation. Morgan Kaufmann, 1997. - 888 с.

13. Sandro Pasquali Mastering Node.js (Community Experience Distilled). - 1 изд. - Birmingham: Packt Publishing, 2013. - 346 с.

14. Simon Holmes Getting MEAN with Mongo, Express, Angular, and Node. - 1 изд. - Shelter Island: Manning Publications, 2015. - 440 с.

15. Srikant Y. N., Shankar P. The Compiler Design Handbook: Optimizations and Machine Code Generation. - 2 изд. - Boca Raton: CRC Press, 2007. - 784 с.

16. Stephen Ludin, Javier Garza Learning HTTP/2: A Practical Guide for Beginners. - 1 изд. - Sebastopol: O'Reilly Media, 2017. - 156 с.

17. Thomas Erl, Ricardo Puttini, Zaigham Mahmood Cloud Computing: Concepts, Technology & Architecture (The Prentice Hall Service Technology Series from Thomas Erl). - 1 изд. - Upper saddle river: Prentice Hall, 2013. - 528 с.

18. Tim Mather, Subra Kumaraswamy, Shahed Latif Cloud Security and Privacy: An Enterprise Perspective on Risks and Compliance (Theory in Practice). - 1 изд. - Sebastopol: O'Reilly Media, 2009. - 338 с.

19. Vasan Subramanian Pro MERN Stack: Full Stack Web App Development with Mongo, Express, React, and Node. - 1 изд. - New York City: Apress, 2017. - 328 с.

20. Yu-Kwong Ricky Kwok Peer-to-Peer Computing: Applications, Architecture, Protocols, and Challenges (Chapman & Hall/CRC Computational Science). - 1 изд. - Boca Raton: CRC Press, 2011. - 216 с.

Электронные ресурсы

21. Top Down Operator Precedence [Электронный ресурс]. – Режим доступа: https://crockford.com/javascript/tdop/tdop.html, свободный. – Загл. С экрана. (дата обращения 16.03.18).

ПРИЛОЖЕНИЕ А. ФАЙЛ КОНФИГУРАЦИИ ДОКЕР КОНТЕЙНЕРА

Содержание Dockerfile

```
version: '3'

services:
  server:
    build: ./
    networks:
      inter-net:
          ipv4_address: 10.5.0.5
    ports:
      - "5000:5000"
    depends_on:
      - mysql

  mysql:
    image: mysql:5.7.22
    ports:
      - "3306:3306"
    volumes:
      - /var/lib/mysql
    environment:
      MYSQL_ROOT_PASSWORD: root
      MYSQL_DATABASE: contract
      MYSQL_USER: root
      MYSQL_PASSWORD: root
    networks:
      - inter-net

networks:
  inter-net:
    ipam:
      driver: bridge
      config:
        - subnet: 10.5.0.0/16
```

ПРИЛОЖЕНИЕ Б. ПРОГРАМНЫЙ КОД ИНТЕРПРЕТАТОРА

Содержание файла inter.js

```
let lex = function(input) {

    // У нас есть несколько разных типов команд-меток (tokens):
    // - операторы: +-*/^%=()
    // - числа, записываемые цифрами: 0123456789
    // - пробелы, которые лексический анализатор будет игнорировать: ' '
    // - идентификатор переменных и функция, записываемых как строки, состоящие из
любых символов, кроме операторов, цифр и пробелов.
    let isOperator = function(c) {
        return /[+\-*\/\^%=(),]/.test(c);
    };
    let isDigit = function(c) {
        return /[0-9]/.test(c);
    };
    let isWhiteSpace = function(c) {
        return /\s/.test(c);
    };
    let isIdentifier = function(c) {
        return typeof c === "string" && !isOperator(c) && !isDigit(c) &&
!isWhiteSpace(c);
    };

    // Мы создадим простой цикл для сканирования текста. Нам понадобится
переменная с именем c,
    // которая будет хранить в себе текущий строковый символ.
    // Также мы создадим функцию с названием advance для перехода вперед на один
строковый символ в тексте
    // и возвращения следующего строкового символа и функцию с названием addToken,
    // которая будет добавлять команду-метку (token) в список команд-меток
(tokens).
    let tokens = [];
    let c;
    let i = 0;

    let advance = function() {
        return c = input[++i];
    };
    let addToken = function(type, value) {
        tokens.push({
            type: type,
            value: value
        });
    };
    while (i < input.length) {
        c = input[i];

        // Если символ в переменной c будет являться пробелом, то мы проигнорируем
его и пойдем разбирать текст дальше.
        // Если символ в переменной c будет являться оператором, то мы добавим
команду-метку (token) оператора в
        // список команд-меток (tokens) и пойдем разбирать текст дальше.
        if (isWhiteSpace(c)) advance();

        else if (isOperator(c)) {
            addToken(c);
```

```
            advance();
        }
        // Для упрощения работы мы определим символ числа, как последовательность
цифр,
        // которые также могут быть разделены точкой (.) для записи десятичных
чисел.
        else if (isDigit(c)) {
            let num = c;
            while (isDigit(advance())) num += c;
            if (c === ".") {
                do num += c;
                while (isDigit(advance()));
            }
            num = parseFloat(num);
            if (!isFinite(num)) throw "Number is too large or too small for a 64-
bit double.";
            addToken("number", num);
        }

        // Все остальные символы мы будем считать идентификаторами переменных,
boolean и функций.
        // Все остальные символы мы будем считать идентификаторами переменных,
boolean и функций.
        else if (isIdentifier(c)) {
            let idn = c;
            while (isIdentifier(advance())) idn += c;

            if (idn === 'true'){
                addToken("boolean", idn);
            }
            if (idn === 'false') {
                addToken("boolean", idn);
            }
            if (idn === 'if'){
                addToken("if", idn);
            }
            if (idn === 'for'){
                addToken("for", idn);
            }
        }

        // И в случае возникновения непредвиденной нами ошибки мы будем
выбрасывать исключение.
        else throw "Unrecognized token.";
    }

    // После того, как мы просканируем текст, мы добавим завершающую команду-метку
(токен) - "end", обозначающую конец.
    addToken("(end)");
    return tokens;
};

let parse = function(tokens) {
    let symbols = {};

    // Нам понадобится некое подобие таблицы символов для ассоциации символа с его
силой присваивания: nud или led. (symbol function)
    // Также нав понадобится функция для ассоциации команды-метки (token) с
соответствующим символом. (interpretToken)
    const symbol = function(id, lbp, nud, led) {
```

```
        if (!symbols[id]) {
            symbols[id] = {
                lbp: lbp,
                nud: nud,
                led: led
            };
        } else {
            if (nud) symbols[id].nud = nud;
            if (led) symbols[id].led = led;
            if (lbp) symbols[id].lbp = lbp;
        }
    };

    // Операторы, которые существуют только в виде разделения или конечной демаркации.
    // Они не являются префиксами или инфиксами, просто добавим их в таблицу символов.
    symbol(",");
    symbol(")");
    symbol("(end)");
    symbol(";");

    let interpretToken = function(token) {
        let F = function() {};
        F.prototype = symbols[token.type];
        let sym = new F;
        sym.type = token.type;
        sym.value = token.value;
        return sym;
    };

    // Способ увидеть, что представляет из себя текущая команда-метка (token) и способ перейти к следующей команде-метке
    let i = 0;
    let token = function() {
        return interpretToken(tokens[i]);
    };
    let advance = function() {
        i++;
        return token();
    };

    // Функция разбирающая синтаксическое дерево по принципу выше.
    let expression = function(rbp) {
        let left;
        let t = token();
        advance();
        if (!t.nud) throw "Unexpected token: " + t.type;
        left = t.nud(t);
        while (rbp < token().lbp) {
            t = token();
            advance();
            if (!t.led) throw "Unexpected token: " + t.type;
            left = t.led(left);
        }
        return left;
    };

    // Инфиксные и префиксные функций, которые мы сможем использовать для определения операторов.
```

```
    const infix = function(id, lbp, rbp, led) {
        rbp = rbp || lbp;
        symbol(id, lbp, null, led ||
            function(left) {
                return {
                    type: id,
                    left: left,
                    right: expression(rbp)
                };
            });
    };
    const prefix = function(id, rbp, nud) {
        symbol(id, null, nud ||
            function() {
                return {
                    type: id,
                    right: expression(rbp)
                };
            });
    };

    prefix("number", 9, function(number) {
        return number;
    });
    prefix("identifier", 9, function(name) {
        if (token().type === "(") {
            let args = [];
            if (tokens[i + 1].type === ")") advance();
            else {
                do {
                    advance();
                    args.push(expression(2));
                } while (token().type === ",");
                if (token().type !== ")") throw "Expected closing parenthesis
')'";
            }
            advance();
            return {
                type: "call",
                args: args,
                name: name.value
            };
        }
        return name;
    });
    prefix("boolean", 9, function(bool) {
        if (bool === 'true'){
            return 1;
        } else {
            return 0;
        }
    });

    prefix("(", 8, function() {
        let value = expression(2);
        if (token().type !== ")") throw "Expected closing parenthesis ')'";
        advance();
        return value;
    });
```

```
    // Определяем все арифметические операторы.
    prefix("-", 7);
    infix("^", 6, 5);
    infix("*", 4);
    infix("/", 4);
    infix("%", 4);
    infix("+", 3);
    infix("-", 3);

    infix("=", 1, 2, function(left) {
        if (left.type === "call") {
            for (let i = 0; i < left.args.length; i++) {
                if (left.args[i].type !== "identifier") throw "Invalid argument
name";
            }
            return {
                type: "function",
                name: left.name,
                args: left.args,
                value: expression(2)
            };
        } else if (left.type === "identifier") {
            return {
                type: "assign",
                name: left.value,
                value: expression(2)
            };
        }
        else throw "Invalid lvalue";
    });

    let parseTree = [];
    while (token().type !== "(end)") {
        parseTree.push(expression(0));
    }
    return parseTree;
};
let evaluate = function(parseTree) {

    // Определяем поведение всех операторов и предварительно определенных
переменных и функций.
    let operators = {
        "+": function(a, b) {
            return a + b;
        },
        "-": function(a, b) {
            if (typeof b === "undefined") return -a;
            return a - b;
        },
        "*": function(a, b) {
            return a * b;
        },
        "/": function(a, b) {
            return a / b;
        },
        "%": function(a, b) {
            return a % b;
        },
        "^": function(a, b) {
            return Math.pow(a, b);
```

```
        },
        ">": function(a, b) {
            return a > b;
        },
        "<": function(a, b) {
            return a < b;
        },
        ">=": function(a, b) {
            return a >= b;
        },
        "<=": function(a, b) {
            return a <= b;
        }
    };

    let letiables = {
        pi: Math.PI,
        e: Math.E
    };

    let functions = {
        sin: Math.sin,
        cos: Math.cos,
        tan: Math.cos,
        asin: Math.asin,
        acos: Math.acos,
        atan: Math.atan,
        abs: Math.abs,
        round: Math.round,
        ceil: Math.ceil,
        floor: Math.floor,
        log: Math.log,
        exp: Math.exp,
        sqrt: Math.sqrt,
        max: Math.max,
        min: Math.min,
        random: Math.random
    };
    let args = {};

    // Функция разбора узлов parseNode рекурсивно проходидит по дереву и исполняет
код.
    let parseNode = function(node) {
        if (node.type === "number") return node.value;
        else if (operators[node.type]) {
            if (node.left) return operators[node.type](parseNode(node.left),
parseNode(node.right));
            return operators[node.type](parseNode(node.right));
        }
        else if (node.type === "identifier") {
            let value = args.hasOwnProperty(node.value) ? args[node.value] :
letiables[node.value];
            if (typeof value === "undefined") throw node.value + " is undefined";
            return value;
        }
        else if (node.type === "assign") {
            letiables[node.name] = parseNode(node.value);
        }
        else if (node.type === "call") {
```

```
            for (let i = 0; i < node.args.length; i++) node.args[i] =
parseNode(node.args[i]);
            return functions[node.name].apply(null, node.args);
        }
        else if (node.type === "function") {
            functions[node.name] = function() {
                for (let i = 0; i < node.args.length; i++) {
                    args[node.args[i].value] = arguments[i];
                }
                let ret = parseNode(node.value);
                args = {};
                return ret;
            };
        }
    };

    let output = "";
    for (let i = 0; i < parseTree.length; i++) {
        let value = parseNode(parseTree[i]);
        if (typeof value !== "undefined") output += value + "\n";
    }
    return output;
};

let calculate = function(input) {
    let tokens = lex(input);
    let parseTree = parse(tokens);
    return evaluate(parseTree);
};


module.exports = calculate;
```