МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ РОССИЙСКОЙ ФЕДЕРАЦИИ
Федеральное государственное автономное образовательное учреждение
высшего образования
**«Южно-Уральский государственный университет
(национальный исследовательский университет)»**
Высшая школа электроники и компьютерных наук
Кафедра системного программирования

ДОПУСТИТЬ К ЗАЩИТЕ

Заведующий кафедрой, д.ф.-м.н.,
профессор

_____ Л.Б. Соколинский

«___»_____ 2024 г.

Разработка языка программирования для Машины Тьюринга

ВЫПУСКНАЯ КВАЛИФИКАЦИОННАЯ РАБОТА
ЮУрГУ – 09.03.04.2024.308-361.ВКР

Научный руководитель,
ст. преподаватель кафедры СП
_____ В.В. Варкентин

Автор работы,
студент группы КЭ-433
_____ М.Н. Масимов

Ученый секретарь
(нормоконтролер)
_____ И.Д. Володченко
«___»_____ 2024 г.

Челябинск, 2024 г.

МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ РОССИЙСКОЙ ФЕДЕРАЦИИ
Федеральное государственное автономное образовательное учреждение
высшего образования
«Южно-Уральский государственный университет
(национальный исследовательский университет)»
Высшая школа электроники и компьютерных наук
Кафедра системного программирования

УТВЕРЖДАЮ
Зав. кафедрой СП
_____ Л.Б. Соколинский
29.01.2024 г.

ЗАДАНИЕ
на выполнение выпускной квалификационной работы бакалавра
студенту группы КЭ-433
Масимову Мураду Намиговичу,
обучающемуся по направлению
09.03.04 «Программная инженерия»

1. **Тема работы** (утверждена приказом ректора от 22.04.2024 г. № 764-13/12)
Разработка языка программирования для Машины Тьюринга.
2. **Срок сдачи студентом законченной работы:** 03.06.2024 г.
3. **Исходные данные к работе**
3.1. Хопкрофт Д., Мотвани Р., Ульман Д. Введение в теорию автоматов, языков и вычислений. – М.: Вильямс, 2002. – 528 с.
3.2. Карпов Ю.Г. Теория автоматов. – Питер, 2003. – 206 с.
4. **Перечень подлежащих разработке вопросов**
4.1. Разработать программный интерфейс для эмуляции машины Тьюринга.
4.2. Реализовать графический интерфейс для эмулятора.
4.3. Спроектировать язык программирования для машины Тьюринга и разработать соответствующий интерпретатор.
5. **Дата выдачи задания:** 29.01.2024 г.

Научный руководитель,
ст. преподаватель кафедры СП В.В. Варкентин

Задание принял к исполнению М.Н. Масимов

ОГЛАВЛЕНИЕ

ВВЕДЕНИЕ

Актуальность

Машина Тьюринга является теоретической моделью вычислений, предложенной Аланом Тьюрингом в 1936 году, и призванной формализовать понятие алгоритма [1]. Она состоит из бесконечной ленты, разделенной на ячейки, каждая из которых может содержать один символ. Машина Тьюринга имеет каретку, которая может перемещаться по ленте, читать и записывать символы в ячейки. Машина также может менять свое состояние, что определяет ее поведение.

Машина Тьюринга является универсальным вычислительным устройством [2], что означает, что она может выполнять любую вычислительную задачу, которую можно выполнить на любом другом компьютере. Однако, Машина Тьюринга является чисто теоретической моделью, и ее реализация на практике в полной мере невозможна, поскольку оперативная память такого устройства, играющая роль бесконечной ленты, будет конечна.

Особую роль Машина Тьюринга играет в таких математических дисциплинах, как теория сложности вычислений и теория вычислимости [3]. На сегодняшний день, в данных областях существуют открытые проблемы. Так, например, гипотеза «$P \neq NP$», до сих пор не была ни доказана, ни опровергнута [4]. Она утверждает, что множество задач класса NP, включающее в себя задачи класса P, не совпадает с множеством задач класса P. Класс NP – это множество таких задач, решение которых возможно проверить за не более чем полиномиальное время, то есть сложность такого алгоритма при размере входных данных n будет определяться как $O(n^k)$ для некоторого конечного k. Класс P – это множество задач, которые возможно решить за полиномиальное время. Поскольку на истинности гипотезы «$P \neq NP$» основана работа большинства криптографических алгоритмов, доказательство гипотезы позволит подтвердить, что такие криптографиче-

ские алгоритмы не могут быть взломаны за разумное время. Опровержение данной гипотезы, напротив, приведет к нарушению криптографической стойкости всех современных алгоритмов шифрования [5].

На сегодняшний день, студенты большинства технических специальностей, связанных с информационными технологиями, изучают Машину Тьюринга на первых курсах обучения [6–10]. Существует множество обучающих эмуляторов Машины Тьюринга, с помощью которых студенты учатся самостоятельно задавать поведение Машины Тьюринга, нужное для решения той или иной задачи. Для этого они задают набор возможных символов для отображения в ленте, набор состояний, в которых может находиться модель, и правила перехода для всех возможных сочетаний состояния машины и символа, считанного кареткой. Очевидно, что такой подход разумно применять только для решения простых задач, не требующих использования большого количества состояний.

Как будет показано далее, процесс описания поведения Машины Тьюринга, для решения даже простейших задач, сопряжен с многократным повторением по сути одних и тех же шаблонных правил, которые можно изложить в более краткой и интуитивно понятной форме. Существование инструмента, позволяющего формулировать эти правила в обобщенном виде, позволило бы сильно упростить процесс создания программ для Машины Тьюринга, а также улучшить понимание алгоритмов, реализованных на Машине Тьюринга.

В данной работе будет разработан язык программирования, позволяющий, с помощью обобщенных шаблонных формулировок, упростить процесс создания программы для Машины Тьюринга.

Постановка задачи

Целью данной выпускной квалификационной работы является разработка языка программирования, позволяющего создавать программы для Машины Тьюринга.

Для достижения поставленной цели необходимо решить следующие задачи:

1) спроектировать язык программирования для Машины Тьюринга и разработать соответствующий интерпретатор;

2) разработать программный интерфейс для эмуляции работы Машины Тьюринга;

3) реализовать графический интерфейс для эмулятора, позволяющий исполнять программы, написанные на проектируемом языке, и визуализировать работу Машины Тьюринга.

Структура и содержание работы

Работа состоит из введения и заключения, четырех глав, списка литературы и двух приложений. Объем работы составляет 68 страниц, объем списка литературы – 23 источника.

В первой главе описываются теоретические сведения о предметной области, и осуществляется обзор существующих аналогов, которые позволяют упростить процесс создания программ для Машины Тьюринга.

Вторая глава посвящена проектированию разрабатываемого языка и его интерпретатора, эмулятора Машины Тьюринга и графического интерфейса для него.

Третья глава описывает подробности программной реализации спроектированной программы и использованные для этого средства разработки.

В четвертой главе представлены результаты тестирования эмулятора и интерпретатора языка.

В приложении А приведены примеры кода на проектируемом языке программирования для тестирования работы разрабатываемого интерпретатора. В приложении Б частично представлен исходный код реализации различных компонентов разрабатываемой программы. В приложении В показан внешний вид аналогов, рассматриваемых в первой главе. В приложении Г приведены диаграммы, составленные на этапе проектирования.

1. АНАЛИЗ ПРЕДМЕТНОЙ ОБЛАСТИ

1.1. Описание предметной области

Устройство Машины Тьюринга

Машина Тьюринга состоит из следующих компонентов [1].

1. Бесконечная лента. Лента разделена на ячейки, каждая из которых может содержать один символ. Символы могут быть из некоторого конечного алфавита. Символ, обозначающий, что ячейка пуста, также по умолчанию входит в этот алфавит.

2. Каретка. Каретка может перемещаться вдоль ленты, читать и изменять символ в ячейке, на которую указывает в данный момент. Каретка также может менять свое состояние, оно же – состояние машины.

3. Таблица переходов. Таблица переходов определяет поведение Машины Тьюринга. Она содержит список всех возможных состояний каретки и символов в ячейке, в которой находится каретка. Для каждого такого состояния и символа, таблица переходов указывает новое состояние каретки, символ, который должен быть записан в ячейку, и направление, в котором должна переместиться каретка (влево, вправо или остаться на месте).

Работа Машины Тьюринга

Машина Тьюринга работает следующим образом [1].

1. Машина находится в начальном состоянии.

2. Каретка считывает символ. Дальнейшие шаги зависят от текущего состояния машины и считанного символа.

3. Каретка записывает новый символ в ячейку.

4. Каретка перемещается в указанном направлении на один шаг, либо остается на месте.

5. Машина переходит в новое состояние.

6. Шаги 2–5 повторяются до тех пор, пока Машина Тьюринга не достигнет конечного состояния.

Пример работы Машины Тьюринга

Предлагается рассмотреть простой пример работы Машины Тьюринга, которая умножает данное десятичное число на два. На рисунке 1 показан пример ленты со схематичным изображением каретки, указывающей на первый символ.

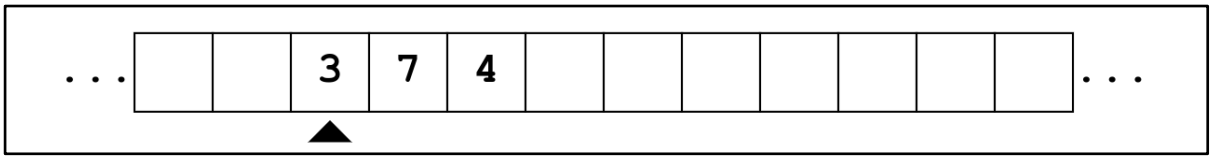

Рисунок 1 – Пример бесконечной ленты

Пусть начальное состояние обозначается как q_1, конечное как q_0, а также определены дополнительные состояния c_0 и c_1. Используемый алфавит состоит из десятичных цифр от 0 до 9, и символа пустой ячейки, обозначенного знаком «_». В таблице 1 приводится таблица переходов, составленная для алгоритма умножения числа в ленте на два.

Таблица 1 – Таблица переходов для алгоритма умножения на два

Символ	**q_1**	**c_0**	**c_1**
_	c_0_L	q_0_R	$q_0$1N
0	$q_1$0R	$c_0$0L	$c_0$1L
1	$q_1$1R	$c_0$2L	$c_0$3L
2	$q_1$2R	$c_0$4L	$c_0$5L
3	$q_1$3R	$c_0$6L	$c_0$7L
4	$q_1$4R	$c_0$8L	$c_0$9L
5	$q_1$5R	$c_1$0L	$c_1$1L
6	$q_1$6R	$c_1$2L	$c_1$3L
7	$q_1$7R	$c_1$4L	$c_1$5L
8	$q_1$8R	$c_1$6L	$c_1$7L
9	$q_1$9R	$c_1$8L	$c_1$9L

Из примера в таблице 1 можно увидеть, что большинство ячеек таблицы заполнены по определенным правилам, описанным далее.

1. В первом столбце, соответствующем начальному состоянию q_1, для всех десятичных цифр действует правило: остаться в состоянии q_1, не переписывать символ в ячейке, переместиться вправо.

2. Во втором столбце, соответствующем состоянию c_0, для всех десятичных цифр действует правило: остаться в состоянии c_0, если цифра меньше 5, иначе перейти в состояние c_1; записать в ячейку остаток от деления на 10 удвоенного значения в текущей ячейке, переместиться влево.

3. В третьем столбце, соответствующем состоянию c_1, для всех десятичных цифр действует правило: перейти в состояние c_0, если цифра меньше 5, иначе остаться в состоянии c_1; записать в ячейку увеличенный на единицу остаток от деления на 10 удвоенного значения в текущей ячейке, переместиться влево.

Также можно заметить сходство состояний c_0 и c_1. Таким образом, правила переходов для данного алгоритма можно выразить, используя обобщения.

1.2. Сравнительный анализ аналогов

Тренажер «Машина Тьюринга» К.Ю. Полякова [11]

Программа доступна для операционной системы Windows. Тренажер представляет собой оконное приложение, с удобным и интуитивно понятным графическим интерфейсом. В верхней части программы отображается лента, по которой можно перемещаться влево и вправо. Правила переходов задаются в таблице в нижней части программы.

Алфавит указывается в специальном текстовом поле. Допускаются только графические символы, знак пробела представляет пустую строку. Также в интерфейсе программы доступны текстовые поля для условия задачи и комментария.

Программа может выполняться непрерывно, с возможностью регулировать скорость исполнения, либо по шагам. Созданную таблицу переходов, вместе с алфавитом, начальным состоянием ленты, комментарием и условием задачи, можно сохранить в файл.

На рисунке 1 приложения В показан интерфейс тренажера «Машина Тьюринга».

Онлайн-симулятор «Turing Machine» [12]

Данное веб-приложение работает в любом современном браузере, и потому доступно на любой операционной системе, что является значительным преимуществом. В интерфейсе данной программы также представлена бесконечная лента, но свободное перемещение по ней недоступно, а каретка всегда находится ровно в середине. Приложение не требует в явном виде указывать алфавит, и позволяет загружать в ленту, а также вводить в таблицу переходов, любые символы Юникода. Сама же программа задается с помощью специального языка программирования в отдельном текстовом поле.

На сайте доступны обучающие материалы, а также несколько примеров алгоритмов, таких как преобразования из десятичной системы в двоичную, копирование строки из бинарных цифр и т. п.

На рисунке 2 приложения В представлен интерфейс веб-приложения.

Выводы по первой главе

В данной главе была описана предметная область. На основе информации из литературы, было приведено определение Машины Тьюринга, и дано краткое описание принципа ее работы. Сформулирована проблема, решение которой предлагается в данной работе. Был проведен сравнительный обзор аналогов, что послужит основой для проектирования эмулятора Машины Тьюринга.

2. ПРОЕКТИРОВАНИЕ

2.1. Проектирование языка программирования

В рамках проектирования языка программирования, будет сформулирована его основная концепция, разработан и формализован синтаксис языка, рассмотрен пример программы на данном языке. Далее будет спроектирована иерархия классов, необходимая для реализации интерпретатора языка.

2.1.1. Концепция

Основная идея проектируемого языка состоит в том, чтобы сделать возможным описывать правила переходов выражениями вида «Для каждого состояния q_i применить набор правил R_i» и «Для каждого символа a_j применить переход T_j». Это значит, что символы и состояния можно определять, используя индексы. Например, символ d_j – может обозначать j-ую десятичную цифру (при $j = 0..9$), а c_i – обобщение для состояний c_0 и c_1. Код на проектируемом языке программирования можно воспринимать как инструкцию по заполнению таблицы переходов.

Для реализации данной идеи предлагается следующее решение. Конечный алфавит и множество состояний, в которых может находиться Машина Тьюринга, задается пользователем явно. Символы и состояния имеют свои наименования в коде. При этом имеется возможность объявить серию символов или состояний. Такие серии, помимо наименования, имеют несколько индексов, каждый из которых может принимать значения в заданном пользователем диапазоне. Например, ранее упомянутый символ d_j имеет один индекс, принимающий значения от 0 до 9.

Благодарю тому, что идентификаторы, обозначающие символы или состояния, могут иметь индексы, становится возможным применение циклов для эффективного заполнения таблицы переходов. Записываемый символ и следующее состояние задаются параметрически, с помощью арифме-

тических операций над индексами, соответствующими текущей итерации. В проектируемом языке программирования предполагается возможность использования таких циклов как для состояний, так и для символов.

Для того, чтобы данный язык программирования можно было эффективно применять для решения реальных задач, необходимо определиться со способом представления входных и выходных данных программы. Входные данные в Машине Тьюринга – это состояние ленты до начала выполнения программы, а выходные данные – состояние ленты после завершения программы. В рамках проектируемого языка программирования будет рассмотрен только текстовый формат данных. Для того, чтобы разрабатываемая система позволяла обрабатывать практически любой набор символов, предлагается использовать стандарт кодирования Юникод [13].

Поскольку конечный алфавит задается пользователем явно, в виде списка идентификаторов, необходимо предусмотреть возможность назначать идентификаторам конкретный набор символов Юникода. Также целесообразно предусмотреть возможность использования идентификаторов, которым не были назначены символы Юникода, в качестве специальных управляющих символов, необходимых для работы алгоритма.

2.1.2. Синтаксис языка

Весь текст исходного кода на проектируемом языке программирования разбит на блоки, состоящие из заголовка и тела блока. Заголовок отделяется от тела блока двоеточием, а тело блока завершается точкой. Блок может описывать используемый в ленте Машины Тьюринга алфавит, возможные состояния машины или поведение машины.

Язык поддерживает однострочные комментарии, начинающиеся со знака «#». Все символы, начиная со знака «#» и заканчивая переносом строки, игнорируются.

В начале необходимо объявить символы, которые могут быть записаны в ленте, и состояния, в которых машина может находиться. Существуют специальные символы и состояния, не требующие объявления. Они обозначаются ключевыми словами: `null` – символ пустой ячейки, `start` – начальное состояние машины, и `end` – конечное состояние.

Для того, чтобы объявить алфавит и состояния, в заголовке блока указываются ключевые слова `A` и `Q` соответственно. В теле таких блоков через запятую перечислены объявления символов или состояний. Блок с объявлением алфавита, так же, как и блок с объявлением состояний, должен присутствовать в коде ровно один раз. После объявления алфавита и состояний, объявляется неограниченное количество блоков, задающих поведение машины.

При объявлении символов (состояний) назначается идентификатор для символа (состояния) или серии символов (состояний). Каждый идентификатор должен быть уникальным. Серия подразумевает, что к идентификатору можно применять индексы в соответствие с формой идентификатора – диапазонами допустимых значений индексов.

Числовые диапазоны и сложные последовательности

Диапазон числовых значений задает последовательность чисел. Диапазон объявляется с помощью специального бинарного оператора «`..`». Например, диапазон `2..6` обозначает возрастающую последовательность чисел от 2 до 6 включительно, а диапазон `3..-1` – убывающую последовательность чисел от 3 до -1 включительно.

Числовые диапазоны и числа можно объединять в сложные последовательности с помощью оператора конкатенации «`&`». Например, выражение «`1..3 & 0 & 4..2 & 5`» обозначает следующую последовательность чисел: 1, 2, 3, 0, 4, 3, 2, 5.

Арифметические выражения

В коде, в тех местах, где ожидается числовое значение, допустимо использовать арифметические выражения. Арифметическое выражение состоит из числовых литералов, имен переменных, унарных и бинарных операторов, а также скобок.

Далее приводится список групп операторов в порядке приоритетов.

1. Возведение в степень (правоассоциативный оператор). Обозначается знаком «^». Операция определена только для неотрицательных значений показателя.

2. Унарный плюс («+») и унарный минус («-»). Унарный минус инвертирует знак выражения, а унарный плюс игнорируется. Допустимо использовать несколько знаков подряд.

3. Мультипликативная группа (левоассоциативная). Доступны операции умножения («*»), деления («/») и взятия остатка от деления («%»).

4. Аддитивная группа (левоассоциативная). Доступны операции сложения («+») и вычитания («-»).

Строковые литералы

Строковые литералы объявляются с помощью одинарных (“'”) либо двойных («"») кавычек. Между кавычками в строковом литерале допускаются любые символы, кроме переноса строки и самого знака кавычки. Знак кавычки должен быть экранирован с помощью обратной косой черты («\»), так же, как и сам символ обратной косой черты, если он подразумевается быть частью строки.

В строковых литералах предусмотрена возможность добавления символа, соответствующего заданному коду в системе кодирования Юникод. Для этого используется управляющая последовательность «\u{…}». В фигурных скобках указывается шестнадцатеричный код символа.

Символьные диапазоны и строки

Оператор «`..`» также применим к строкам длины 1, что позволяет объявить символьный диапазон. Символьный диапазон описывает строку символов Юникода. Например, диапазон `'1'..'5'` равносилен строковому литералу `'12345'`, а диапазон `'f'..'a'` равносилен записи `'fedcba'`.

Строковые диапазоны и литералы можно объединять в новые строки с помощью оператора конкатенации «`&`». Например, выражение «`'f'..'a' & '_' & 'hello' & '1'..'3' & '!'`» равносильно строковому литералу `'fedcba_hello123!'`.

Объявления символов

Объявления символов могут иметь следующий вид.

1. Явное объявление `null`. Символ `null` можно, хотя это и не требуется, объявить частью алфавита явно. При этом, можно присвоить символу `null` текстовое представление, т.е. символ Юникода, например: `null = '_'`. По умолчанию, символу `null` соответствует код U+0. Допускается использовать такое объявление не более одного раза.

2. Объявление символа или серии символов. Разрешается присваивать символу или серии символов их текстовое представление, задаваемое строкой, например допустима такая запись: `foo = 'X'`, `bar[0..1][1..3] = 'a'..'f'`. Таким образом foo обозначает символ «X», а ссылками `bar[0][1]`, `bar[0][2]`, `bar[0][3]`, `bar[1][1]`, `bar[1][2]` и `bar[1][3]` обозначаются, соответственно, символы «a», «b», «c», «d», «e» и «f». По умолчанию символы не имеют текстового представления.

3. Присвоение значения. Если ранее была объявлена серия символов, которой не была присвоена строка с их текстовым представлением, то разрешается задать текстовое представление отдельному символу из серии. Также разрешается присваивать символу значение `null`. Например: `bar[0..1][1..3], bar[0][3] = null, bar[1][2] = 'X'`. Таким обра-

зом, ссылкой `bar[0][3]` обозначается тот же символ, который обозначается ключевым словом `null`, а `bar[1][2]` обозначает символ «X».

Объявления состояний

Объявления состояний могут иметь следующий вид.

1. Явное объявление `start` и `end`. Состояния `start` и `end` не требуется, но разрешается объявлять явно. Такое объявление не имеет никакого эффекта.

2. Объявление состояния или серии состояний. Например: `foo, bar[0..1][1..3]`.

3. Присвоение специального состояния. Если ранее была объявлена серия состояний, то разрешается назначить отдельному состоянию из серии роль одного из двух специальных состояний – `start` или `end`. Например: `q[1..5], q[1] = start, q[5] = end`. Таким образом, ссылкой `q[1]` обозначается начальное состояние машины, а `q[5]` обозначает конечное состояние.

Итерируемая ссылка

Итерируемая ссылка – это выражение, описывающее последовательность ссылок на состояние или символ. Итерируемая ссылка задает цикл, с каждой итерацией которого меняются индексы, применяемые к идентификатору. Значениям индексов можно назначить имена переменных. Эти переменные можно использовать в теле цикла в арифметических выражениях. Предусмотрена возможность объявить анонимную переменную, указав вместо ее имени символ «_». Последовательность значений, которые принимает индекс в цикле, по умолчанию совпадает с диапазоном значений, заданным при объявлении серии. Однако предусмотрена возможность задать свою последовательность значений.

Предлагается рассмотреть следующий пример. Пусть объявлена серия `q[0..1][0..9][3..1]`. Тогда итерируемая ссылка «`q{x}{y | 1..2 & 6..5 & 3}[2]`» описывает следующую последовательность ссылок:

q[0][1][2], q[0][2][2], q[0][6][2], q[0][5][2], q[0][3][2], q[1][1][2], q[1][2][2], q[1][6][2], q[1][5][2], q[1][3][2]. Здесь x и y – переменные, хранящие значения первого и второго индексов.

Определение поведения состояний

Для того, чтобы задать поведение машины, в заголовке блока размещается итерируемая ссылка на состояние (либо ключевое слово start, если нужно определить поведение машины в начальном состоянии), а в теле блока – список правил перехода для текущего состояния, разделенных точкой с запятой. Для каждого состояния из последовательности, заданной в заголовке блока, применяются указанные в теле блока (являющегося и телом цикла, заданного итерируемой ссылкой) правила перехода.

Правила переходов

Правила переходов применяются к отдельным символам или последовательностям символов. Для каждого символа из последовательности производится запись в соответствующую ячейку таблицы переходов.

Правила перехода описываются с помощью знака «->». Слева от знака «->» указывается считываемый символ. Он задается с помощью итерируемой ссылки, либо ключевым словом null. Справа от знака «->» через запятую указываются, слева направо, записываемый символ, направление перемещения каретки и следующее состояние машины. Направление перемещения каретки задается одним из трех ключевых слов: L, R либо N, означающих, соответственно, «влево», «вправо» и «на месте».

Предлагается рассмотреть следующий пример. Пусть объявлены серии символов a[1..2] и b[0..5], и серия состояний q[1..2]. Тогда правило переходов «a{n} -> b[2 * n + 1], R, q[n]» означает следующее. Если считан символ a[1], то в ленту будет записан символ b[3], а машина перейдет в состояние q[1]. Если же считан символ a[2], то в ленту будет записан символ b[5], а следующим состоянием будет q[2]. Во всех случаях, каретка будет перемещена вправо.

На месте записываемого символа и следующего состояния допускается использовать ключевое слово `same`, означающее, что символ в ячейке ленты или состояние машины не меняются.

Пример программы

В листинге 1 представлен пример программы, написанной на проектируемом языке программирования. Данная программа принимает на вход неотрицательное целое десятичное число, и возвращает его удвоенное значение на выходе.

Листинг 1 – Пример программы на проектируемом языке

```
A: dec[0..9] = '0'..'9'.
Q: mul[0..1].

start: # Начальное состояние
  dec{_} -> same, R, same;
  null -> same, L, mul[0].

mul{c}: # Состояния mul[0] и mul[1]
  dec{n | 0..4} -> dec[n * 2 + c], L, mul[0];
  dec{n | 5..9} -> dec[n * 2 + c - 10], L, mul[1].

mul[0]: null -> same, R, end.
mul[1]: null -> dec[1], N, end.
```

Формализация синтаксиса

В листинге 2 представлено формальное описание генеративных правил проектируемого языка программирования в расширенной форме Бэкуса-Наура [14].

Листинг 2 – Формализация синтаксиса языка

```
<исходный_код> ::= ((<состояния> <алфавит>) | (<алфавит> <состояния>))
{<опр_сост>}.
<состояния> ::= "Q" ":" <объявл_сост_расш> { "," <объявл_сост_расш> } ".".
<объявл_сост_расш> ::= "start" | "end" | <объявл_диап> | <назн_сост>.
<назн_сост> ::= <ссылка> "=" ("start" | "end").
<алфавит> ::= "A" ":" <объявл_симв_расш> { "," <объявл_симв_расш> } ".".
<объявл_симв_расш> ::= ("null" ["=" <строка>]) | <объявл_диап_симв> |
<назн_симв>.
<объявл_диап_симв> ::= <объявл_диап> ["=" <строка_расш>].
<строка_расш> ::= [ <строка_расш> "&" ] (<строка> | <симв_диап>)
<назн_симв> ::= <ссылка> "=" {"null" | <строка>}.
<строка> ::= "'" {<символ>} "'" | '"' {<символ>} '"'
<симв_диап> ::= <строка> ".." <строка>
<символ> ::= "\\" | "\'" | '\"' | ("\u{" <число_шестн> "}") |
<символ_юникода>.
<объявл_диап> ::= <идент> { "[" <числ_диап> "]" }.
<числ_диап> ::= <выражение> ".." <выражение>.
```

```
<выражение> ::= <выр_слаг> { ("+" | "-") <выр_слаг> }.
<выр_слаг> ::= <выр_множ> { ("*" | "/" | "%") <выр_множ> }.
<выр_множ> ::= { "+" | "-" } <выр_элем> { "^" { "+" | "-" } <выр_элем> }.
<выр_элем> ::= <число> | "(" <выражение> ")" | <идент>.
<ссыл_обр> ::= "[" <выражение> "]".
<ссыл_цикл> ::= "{" (<идент> | "_") ["|" <числ_диап_расш>] "}".
<числ_диап_расш> ::= [<числ_диап_расш> "&"] (<выражение> | <числ_диап>).
<ссылка> ::= <идент> { <ссыл_обр> }.
<ссылка_итер> ::= <идент> { <ссыл_обр> | <ссыл_цикл> }.
<опр_сост> ::= ("start" | <ссылка_итер>) ":" <правило> {";" <правило>} ".".
<правило> ::= ("null" | <ссылка_итер>) "->" <ссылка_симв> "," <напр> ","
<ссылка_сост>.
<ссылка_симв> ::= <ссылка> | "null" | "same".
<напр> ::= "N" | "L" | "R".
<ссылка_сост> ::= <ссылка> | "start" | "end" | "same".
<идент> ::= <буква> { <буква> | <цифра> | "_" }
<цифра> ::= "0" | ... | "9".
<буква> ::= "a" | ... | 'z' | 'A' | ... | 'Z'.
<цифра_шестн> ::= <цифра> | "a" | ... | "f" | "A" | ... | "F".
<число> ::= <цифра> {<цифра>}.
<число_шестн> ::= <цифра_шестн> {<цифра_шестн>}.
```

2.1.3. Интерпретатор

Основными компонентами интерпретатора языка программирования [15] являются лексический анализатор (токенизатор), разбивающий исходный код программы на лексемы (токены), и синтаксический анализатор (парсер), объединяющий токены в группы в соответствии с грамматикой языка. Результатом работы интерпретатора является объект, описывающий дерево выражений, из которых состоит исходный код программы, который далее может быть использован для трансляции программы в исполняемый вид.

Лексический анализатор

Лексический анализ – это процедура, в ходе которой интерпретатор последовательно считывает символы исходного кода, формируя их в примитивные единицы языка, такие как ключевые слова, идентификаторы, литералы и т.д. На рисунке 3 приложения Г приведена диаграмма деятельности [16] проектируемого лексического анализатора.

Работа проектируемого токенизатора начинается с инициализации флага `is_comment`, означающего, что был считан символ начала однострочного комментария, и все последующие символы в строке должны

быть проигнорированы. Далее основной цикл начинается с того, что пропускаются следующие пробельные символы и однострочный комментарий, если установлен флаг `is_comment`. Затем, если следующий символ – это знак однострочного комментария, устанавливается флаг `is_comment`, и происходит возврат в начало основного цикла. Наконец, токенизатор последовательно пытается считать одну из следующих лексем: знак препинания, идентификатор или ключевое слово, строковый или числовой литерал. Если ни одну из возможных лексем не удается считать, генерируется ошибка токенизации, и выполнение процедуры завершается. Основной цикл завершается, когда все символы в исходном коде были обработаны, и по завершению цикла процедура возвращает готовую цепочку лексем.

Синтаксический анализатор

Для реализации синтаксического анализатора необходимо спроектировать иерархию классов, отвечающих за хранение информации о сложных компонентах языка, таких как числовые последовательности, составные строки, итерируемые ссылки и т.д. На рисунке 4 приложения Г приведена диаграмма классов [16] проектируемого синтаксического анализатора.

Главным классом в иерархии классов синтаксического анализатора является класс `Parser`. Он хранит всю информацию, необходимую для трансляции кода в исполняемый вид. В частности, класс `Parser` содержит в себе по одному объекту классов `Alphabet` и `States`, список объектов типа `StateBlock`, и объект типа `Context`.

Классы `Alphabet` и `States` хранят информацию об объявленных символах и состояниях соответственно. Для того, чтобы назначить каждому отдельному состоянию и символу (за исключением символов, которым назначены конкретные символы Юникода) числовые идентификаторы, используется класс `IdSpace`. Для символов, которым назначены знаки Юникода, используется класс `SymSpace`.

Для описания отдельных идентификаторов в классе IdSpace используются объекты класса IdDesc, который содержит имя символа и его размерность – последовательность числовых диапазонов (idxrange_t), задающих значения индексов, применимые к данному идентификатору. Класс SymDesc, используемый аналогичным образом в классе SymSpace, является классом-наследником IdDesc. Помимо информации, содержащейся в базовом классе, SymDesc хранит объект класса StringCat, описывающего набор символов, назначаемых данной серии символов.

Класс idxrange_t является классом шаблона Range<T>. Шаблон класса Range<T> является обобщением для диапазона значений произвольного типа T. Он нужен, поскольку в проектируемом языке, кроме числовых диапазонов, также имеют место символьные диапазоны – symrange_t. Класс Range<T> также реализует интерфейс IArray<T>. Эта абстракция позволяет применять один и тот же метод для обращения к элементам объектов разных типов.

Класс StringCat является способом описать конкатенацию строк. Объект данного класса хранит список объектов, реализующих интерфейс istring_t, который является классом шаблона IArray<T>. Интерфейс istring_t реализуют класс symrange_t, упомянутый ранее, а также класс StringValue, являющийся оберткой для строкового литерала. StringCat также реализует интерфейс istring_t.

Класс Context отвечает за хранение значений переменных, которые могут быть использованы в выражениях. Также он отвечает за хранение списка всех используемых имен для того, чтобы гарантировать их уникальность.

Класс StateBlock состоит из объекта класса IdRefIterEval, представляющего итерируемую ссылку на состояние, и списка объектов класса Rule, описывающего правила перехода. В объекте класса StateBlock хранится не менее одного правила, в силу требований синтаксиса языка. Класс

Rule хранит объект класса `IdRefIterEval`, описывающий итерируемую ссылку на считываемый символ, два объекта класса `IdRefEval`, описывающих ссылку на записываемый символ и следующее состояние, и направление движения каретки.

Ссылка, описываемая классом `IdRefEval`, содержит последовательность выражений, задающих индексы при данном идентификаторе. Поскольку эти выражения могут зависеть от переменных, изменяющих свои значения, объект этого класса хранит именно выражения, а не числовые значения. Эти выражения описываются абстрактным классом `IEvaluable`. Для получения ссылки с конкретными значениями индексов, на основе объекта класса `IdRefEval` создается объект класса `IdRef`.

Итерируемая ссылка, описываемая классом `IdRefIterEval`, содержит в себе последовательность вложенных циклов, каждый из которых пробегает по заданным значениям некоторой переменной, которая определяет значение одного из индексов при идентификаторе. Эти циклы описываются классом `IndexIterEval` При этом некоторые индексы могут не изменять свои значения, и аналогично индексам в классе `IdRefEval`, описываются классом `IEvaluable`. Для того, чтобы можно было запустить цикл, в каждой итерации которого будет вычисляться следующая ссылка `IdRef`, на основе класса `IdRefIterEval` создается объект класса `IdRefIter`. Это необходимо не только для того, чтобы вычислить значения фиксированных индексов, но и для того, чтобы вычислить последовательности значений, которые меняются в циклах, поскольку эти последовательности также могут определяться некоторыми выражениями.

Класс `IndexIterEval`, описывающий цикл, в рамках которого меняется значение одной переменной, помимо имени переменной, хранит последовательность значений, которые будет принимать эта переменная. Эта последовательность задается как конкатенация числовых диапазонов. Для описания такой последовательности используется класс `IdxRangeCatEval`.

По аналогии с классом `IdRefIterEval`, чтобы запустить цикл, с каждой итерацией которого будет меняться значение переменной, на основе объекта класса `IndexIterEval` создается объект класса `IndexIter`.

Класс `IndexIter` хранит последовательность значений, которые должна принимать переменная, в виде объекта класса `IdxRangeCat`. Объект класса `IdxRangeCat` создается на основе класса `IdxRangeCatEval`, и хранит конкатенацию фиксированных числовые диапазонов. При создании объекта `IndexIter`, внутри него создается объект вспомогательного класса `ctx::Variable`. Класс `ctx::Variable` устроен таким образом, что при создании объекта, соответствующая переменная будет автоматически создана и помещена в контекст (объект класса `Context`). При уничтожении объекта класса `ctx::Variable` переменная удаляется из контекста.

Как было упомянуто ранее, математические выражения, используемые в вышеописанных классах, реализуют интерфейс `IEvaluable`. Классы `Number` и `math::Variable` – это примитивы, реализующие интерфейс `IEvaluable`. Из них составляются сложные выражения с помощью класса `Expression`, также реализующего интерфейс `IEvaluable`. Он объединяет в себе два объекта `IEvaluable` и бинарный оператор, применяемый к ним. Таким образом, объект класса `Expression` имеет рекурсивную структуру.

2.2. Проектирование эмулятора Машины Тьюринга

Далее сформулированы функциональные и нефункциональные требования к проектируемой программе-эмулятору Машины Тьюринга, спроектирован программный интерфейс для эмуляции Машины Тьюринга, и создан макет пользовательского графического интерфейса, представляющего из себя оконное приложение.

2.2.1. Требования к программе

Функциональные требования

Функциональные требования – это требования, которые определяют действия, которые должна выполнять система, без учета ограничений, связанных с ее реализацией, то есть определяют поведение системы в процессе обработки информации [17]. Разрабатываемая программа должна удовлетворять следующим функциональным требованиям:

- позволять создавать, открывать, редактировать и сохранять файлы исходного кода;
- позволять загружать данные в ленту из файла;
- позволять сохранять в файл текущее состояние ленты;
- позволять изменять данные в ленте через диалоговое окно;
- отображать актуальное состояние ленты в окрестности текущего положения каретки;
- позволять перемещать каретку вручную;
- отображать текущее состояние Машины Тьюринга;
- позволять запускать выполнение программы с заданной скоростью и возможностью приостановить работу;
- позволять ручное пошаговое выполнение программы;
- позволять восстанавливать начальное состояние машины и ленты.

Нефункциональные требования

Нефункциональные требования – это требования, которые не определяют поведение системы, но описывают атрибуты системы или атрибуты системного окружения [17].

Разрабатываемая программа должна работать на операционной системе Windows. Эмулятор должен иметь понятный графический интерфейс. Все надписи в программе должны быть переведены на русский язык. Должна быть доступна справочная информация.

2.2.2. Программный интерфейс

Программный интерфейс эмулятора – это модуль, отвечающий непосредственно за работу Машины Тьюринга, за загрузку и выгрузку состояния ленты, и за загрузку программы на проектируемом языке программирования в эмулятор. Диаграмма классов программного интерфейса эмулятора приведена на рисунке 2.

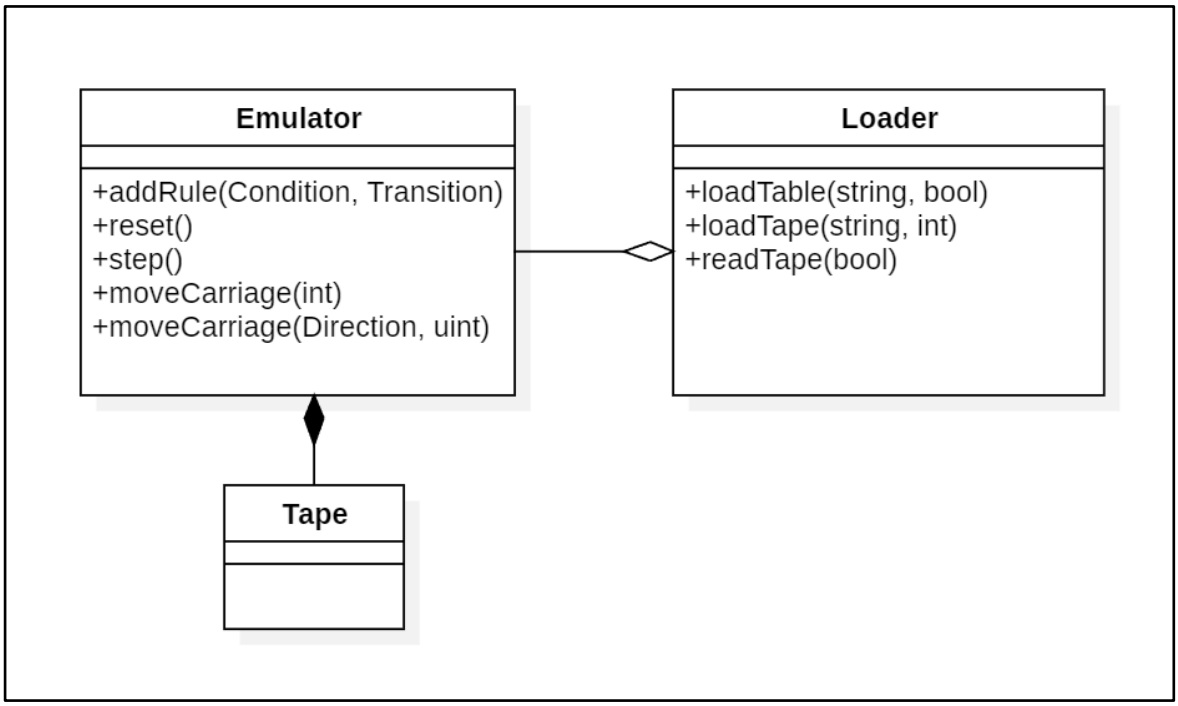

Рисунок 2 – Диаграмма классов программного интерфейса эмулятора

Класс `Emulator` хранит таблицу переходов, текущее состояние машины и объект класса `Tape`. Класс `Tape` хранит состояние ленты, включая положение каретки. Отдельные ячейки таблицы переходов заполняются с помощью метода `addRule`. Переход в следующее состояние производится методом `step`. Метод `reset` сбрасывает состояние машины в начальное. Также есть возможность передвинуть каретку в нужном направлении на заданное количество шагов с помощью метода `moveCarriage`.

За загрузку программы в объект класса `Emulator` отвечает метод `loadTable` класса `Loader`. С помощью того же класса производится загрузка и выгрузка данных в ленте, вызовами методов `loadTape` и `readTape`.

Объект класса `Loader` хранит ссылку на объект класса `Emulator`. С этим объектом и будут производится операции загрузки таблицы переходов, а также чтение и запись данных в ленте.

2.2.3. Графический интерфейс

Графический интерфейс эмулятора представляет собой окно, в котором отображается полоска меню, часть бесконечной ленты Машины Тьюринга, кнопки управления эмулятором, область для редактирования текста и строка состояния программы.

В полоске меню, расположенной в верхней части окна, есть меню «Файл» и «Лента», а также кнопка «Справка». Через меню «Файл» можно открыть, сохранить или создать новый файл, а также доступна кнопка «Выход». Меню «Лента» позволяет загрузить или выгрузить в файл данные в ленте, либо открыть диалоговое окно для изменения данных в ленте. Кнопка «Справка» открывает окно с руководством пользователя.

Под полоской меню расположен элемент, в котором частично отображается лента Машины Тьюринга. Количество отображаемых ячеек должно адаптироваться под ширину окна. Положение каретки фиксировано посередине, то есть при перемещении каретки по ленте, визуально будет двигаться лента. Должна быть также предусмотрена возможность перемещать каретку вручную, нажатием по нужной ячейке.

Под лентой находится панель управления, в которой доступны различные элементы управления программой, а также отображается текущее состояние машины.

Основную часть окна занимает текстовое поле, предназначенное для ввода программы на проектируемом языке программирования. В нижней части окна, в строке состояния, отображается имя файла, если файл был открыт. Наличие звездочки возле имени файла сигнализирует о том, что

изменения, внесенные в текст, не были сохранены. Также в строке состояния указывается положение текстового курсора – номера строки и столбца.

На рисунке 3 изображен макет графического интерфейса эмулятора.

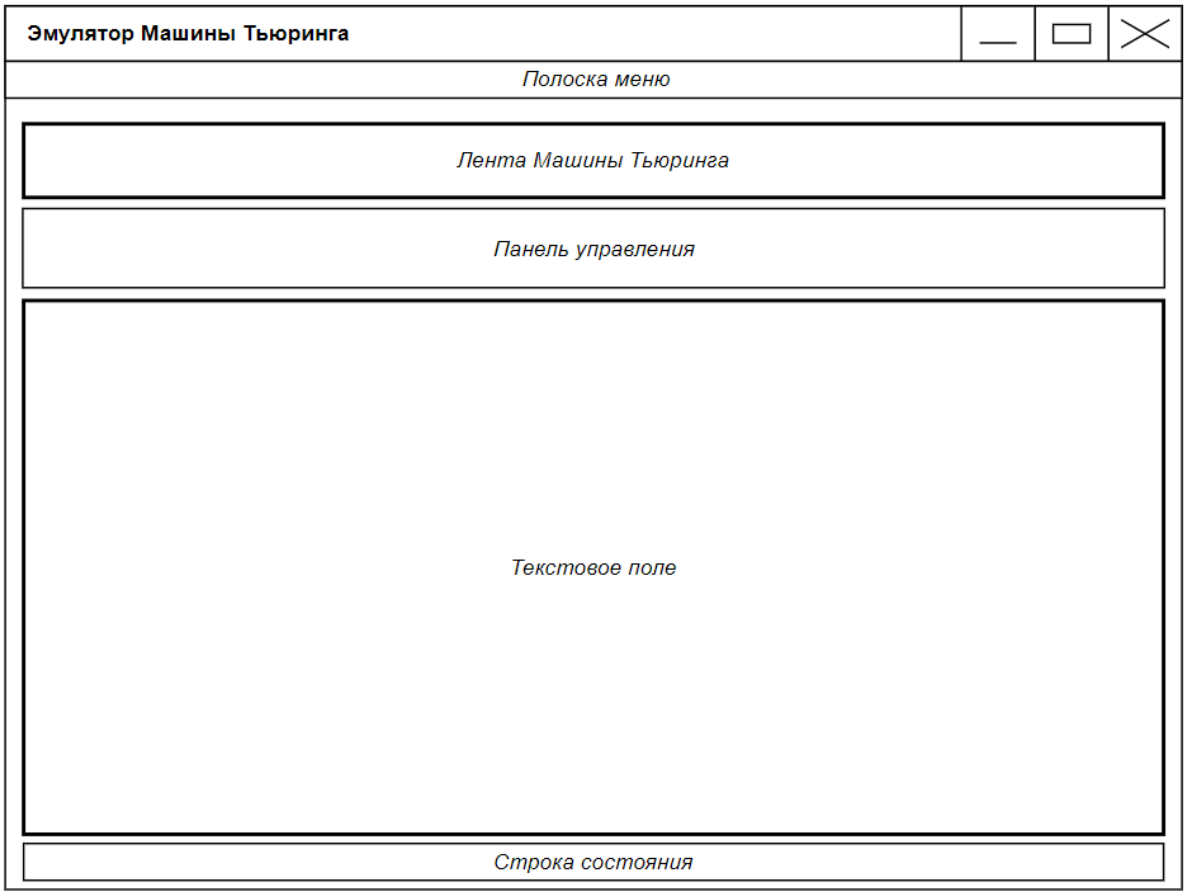

Рисунок 3 – Макет графического интерфейса

Выводы по второй главе

В этой главе был описан и формализован синтаксис проектируемого языка программирования. Был спроектирован интерпретатор языка программирования. Для этого была построена диаграмма деятельности для лексического анализатора, а также диаграмма классов для синтаксического анализатора. Кроме того, был спроектирован программный интерфейс эмулятора Машины Тьюринга, и составлен макет графического интерфейса эмулятора.

3. РЕАЛИЗАЦИЯ

Для реализации языка программирования и эмулятора использованы следующие технологии, перечисленные ниже.

1. Язык программирования C++ [18] (стандарт языка – C++20).
2. Фреймворк QT 6.6.2 [19].
3. Модуль QtWidgets для реализации графического интерфейса [20].
4. Среда разработки QtCreator 12.0 [21].
5. Собрание библиотек Boost 1.85.0 [22].
6. Система автоматизации сборки CMake 3.28.3 [23].

3.1. Реализация интерпретатора языка программирования

Для того, чтобы привести код программы в исполняемый вид, необходимо проанализировать текст исходного кода, объединив последовательности символов в лексемы (лексический анализ), которые затем будут объединены в более сложные структуры (синтаксический анализ). Далее приводятся подробности реализации необходимых компонентов интерпретатора.

3.1.1. Лексический анализатор

Лексический анализ будет осуществляться отдельной функцией, принимающей на вход текст исходного кода, и возвращающей цепочку лексем. Эта функция, и вспомогательные процедуры для ее работы, обернуты в отдельный класс `Tokenizer`. Реализация лексического анализатора представлена в листинге 14 приложения Б.

Структура `Token`, описывающая лексемы, из которых и состоит конечная цепочка, содержит в себе информацию о его типе, значении (если применимо), а также расположение токена в исходном коде – номера строки и столбца. Последнее необходимо для того, чтобы при генерации ошибок синтаксического анализа была возможность сообщить пользователю

не только то, в чем заключается ошибка, но и где в исходном коде она была допущена. В листинге 3 приводится определение структуры `Token`.

Листинг 3 – Структура `Token`

```
struct SourceRef {
    long row, col;
};

struct Token {
    enum Type { NONE, KW_A, KW_Q, KW_NULL, KW_START, KW_END, KW_N, KW_L,
KW_R, KW_SAME, COLON, SEMICOLON, COMMA, PERIOD, ARROW, ASSIGN, RANGE, CAT,
ANON, ITER, BRACKET_L, BRACKET_R, BRACE_L, BRACE_R, PAR_L, PAR_R, PLUS,
MINUS, MUL, DIV, MOD, POW, ID, NUMBER, STRING } type;

    SourceRef srcRef;
    QVariant value;
};
```

Токенизатор использует вспомогательные функции. Одна из них, функция `nextName`, производит попытку вычленения имени – идентификатора либо ключевого слова. Если имя не удается считать, функция возвращает пустую строку, сигнализируя о неудаче. Имя может состоять из букв латинского алфавита, цифр и символа нижнего подчеркивания, но всегда должно начинаться с буквы. В листинге 4 приведена реализация функции `nextName`.

Листинг 4 – Реализация функции `nextName`

```
QString Tokenizer::nextName(QStringView tail, QString::const_iterator &it)
{
    static QRegularExpression re(R"raw(^[a-zA-Z][_0-9a-zA-Z]*)raw");

    auto match = re.matchView(tail);
    if (match.hasMatch()) {
        auto ret = match.captured();
        it += ret.size();
        return ret;
    }

    return QString();
}
```

Следующая вспомогательная процедура – функция `nextLiteral`. Она пытается извлечь следующий числовой либо строковый литерал. Строковый литерал может содержать экранированные символы и управляющие последовательности. Данная функция ответственна за то, чтобы

обработать эти специальные последовательности, и вернуть на выходе токен с готовой строкой. В листинге 15 приложения Б приведена реализация процедуры `nextLiteral`.

3.1.2. Синтаксический анализатор

Синтаксический анализ осуществляется с помощью иерархии классов, каждый из которых отвечает за работу отдельной сущности в языке. Основным классом в этой иерархии является класс `Parser`. В листинге 5 представлена реализация конструктора класса `Parser`, производящего синтаксический анализ цепочки токенов.

Листинг 5 – Реализация основной процедуры синтаксического анализа

```
Parser::Parser(QList<Token>::const_iterator begin,
QList<Token>::const_iterator end) {
    QList<Token>::const_iterator it = begin, it_period;

    while (it != end) {
        it_period = nextToken(it, end, Token::PERIOD);
        if (it_period == end) throw ParseError{ CommonError{
            .srcRef = end->srcRef, .msg = QObject::tr("Expected '.'") }};

        switch (it->type) {
        case Token::KW_A:
            if (alph) throw ParseError{ CommonError{
                .srcRef = end->srcRef, .msg = QObject::tr("Alphabet has
already been declared before") }};
            ++it;
            if (it->type != Token::COLON) throw ParseError{ CommonError{
                .srcRef = end->srcRef, .msg = QObject::tr("Expected ':'")
}};
            ++it;
            this->alph = std::make_shared<Alphabet>(it, it_period, this-
>context);
            break;
        case Token::KW_Q:
            if (states) throw ParseError{ CommonError{
                .srcRef = end->srcRef, .msg = QObject::tr("States have
already been declared before") }};
            ++it;
            if (it->type != Token::COLON) throw ParseError{ CommonError{
                .srcRef = end->srcRef, .msg = QObject::tr("Expected ':'")
}};
            ++it;
            this->states = std::make_shared<States>(it, it_period, this-
>context);
            break;
        default:
            if (!this->alph || !this->states) throw ParseError{
CommonError{
                .srcRef = it->srcRef, .msg = QObject::tr("Alphabet and
states must be declared at the beginning") }};
```

```
            StateBlock block(it, it_period);
            this->blocks.push_back(std::move(block));
        }

        it = it_period;
        ++it;
    }

    if (!this->alph) throw ParseError{ CommonError{
        .srcRef = end->srcRef, .msg = QObject::tr("Alphabet has not been
declared") }};
    if (!this->states) throw ParseError{ CommonError{
        .srcRef = end->srcRef, .msg = QObject::tr("States have not been
declared") }};
}
```

Конструктор класса `Parser` работает следующим образом. Пока цепочка токенов не обработана полностью, производится поиск положения в цепочке следующего токена «точка», завершающего блок объявления в коде. Если такой токен не был найден, происходит ошибка парсинга. Далее, по типу текущего токена, определяется вид блока: объявление символов (начинается с ключевого слова «`A`»), объявление состояний (ключевое слово «`Q`»), либо определение поведения состояний. За парсинг каждого из перечисленных видов блока отвечает свой класс: `Alphabet`, `States` и `StateBlock` соответственно.

В конструкторе класса `Parser` производятся дополнительные проверки на соответствие исходного кода определенным требованиям. Алфавит и состояния должны быть объявлены в коде ровно один раз, причем до первого появления блока, определяющего поведение состояний.

Класс `Parser` так же создает объект класса `Context`. Этот объект отвечает за хранение множества используемых в программе имен, чтобы обеспечить уникальность каждого идентификатора. Кроме того, класс `Context` используется для хранения значений локальных переменных, которые существуют, когда исполняется цикл, запущенный итерируемой ссылкой. Объектом класса `Context` владеет класс `Parser`, и передает его остальным классам по ссылке.

Алфавит и состояния

Классы `Alphabet` и `States` обрабатывают последовательность объявлений символов и состояний, сохраняя информацию о них. Для хранения этих данных используются классы `IdSpace` и `SymSpace`.

Класс `IdSpace` назначает символам или состояниям уникальные номера – 32-битные беззнаковые целочисленные идентификаторы. В качестве параметра задается минимальное значение номера, который может быть назначен символу или состоянию. Это необходимо для того, чтобы отличать назначенные номера от специальных номеров. Также класс позволяет назначить отдельным символам или состояниям, которые уже добавлены в объект, специальный номер.

Для реализации класса `IdSpace` используется два ассоциативных списка – упорядоченный и неупорядоченный. В первом из них, ключом является начальный номер, относящийся к данному идентификатору, а значением – структура `IdDesc`, состоящая из имени идентификатора и диапазонов индексов, задающих размерность серии. Второй ассоциативный список задает обратное отношение: ключом является имя идентификатора, а значением – начальный номер. Количество занимаемых идентификатором номеров равно количеству символов или состояний, описываемых данным идентификатором. Также используется еще один неупорядоченный ассоциативный список для присваивания отдельным символам или состояниям специального номера. Таким образом, обеспечивается возможность перехода от ссылки на состояние или символ к соответствующему номеру, и наоборот, от назначенного номера к ссылке. В листинге 16 приложения Б приведена реализация основных методов класса `IdSpace`.

Класс `SymSpace` используется для символов и серий символов, которым изначально назначены текстовые представления, т.е. символам, не требующим назначения уникальных номеров. Символы в текстовом представлении также представлены 32-битными значениями – кодами симво-

лов в системе кодирования UTF-32, также известной как UCS-4 [13]. Для реализации класса `SymSpace` был использован неупорядоченный ассоциативный список, где ключом является имя идентификатора, а значением – структура `SymDesc`. Структура `SymDesc` включает в себя информацию из ранее упомянутой структуры `IdDesc`, но также содержит текстовое представление символов. В листинге 17 приложения Б приведена реализация основных методов класса `SymSpace`.

В классе `States` используется только объект класса `IdSpace`. Минимальное значение назначаемого номера равно 2. Номерами 0 и 1 обозначаются, соответственно, конечное и начальное состояния машины. В листинге 18 приложения Б приведена реализация синтаксического анализа объявлений состояний.

Класс `Alphabet` использует для хранения информации о символах не только `IdSpace`, но и `SymSpace`. Специальными номерами считаются те числа, в двоичном представлении которых старший бит равен нулю. Такое решение обосновано тем, что стандарт UCS-4 определяет 31-битную форму кодировки, в которой используются значения от 0 до $7FFFFFFF_{16}$ [13]. Следовательно, в `IdSpace` минимальным назначаемым номером является значение 80000000_{16}. В листинге 19 приложения Б приведена реализация синтаксического анализа объявлений символов.

Диапазоны и последовательности

Для представления числовых и символьных диапазонов используется шаблон класса `Range<T>`. Здесь `T` – тип значения в диапазоне. Для числовых диапазонов используется тип `index_t`, а для символьных – `sym_t`. Тип числового диапазона обозначается как `idxrange_t`, а символьного – как `symrange_t`. Классы шаблона `Range<T>` также наследуют интерфейс `IArray<T>`, предоставляющий возможность обращения к произвольному элементу некоторой последовательности и определения размера этой по-

следовательности. Этот интерфейс будет использован для реализации строковых типов.

Конкатенация числовых диапазонов хранится в объекте класса `IdxRangeCat` в виде списка объектов типа `idxrange_t`. Однако, поскольку числовые диапазоны могут быть заданы в коде не только числами, но и выражениями, необходимы отдельные классы, представляющие числовые диапазоны и последовательности в их исходном виде. Для этого используются классы `IdxRangeCatEval` и `IdxRangeEval`, описывающие, соответственно, числовую последовательность и числовой диапазон.

Класс `IdxRangeCatEval`, аналогично классу `IdxRangeCat`, хранит список объектов класса `IdxRangeEval`. Класс `IdxRangeEval` представлен, как два выражения типа `IEvaluable`, значения которых определяют границы диапазона. С помощью метода `eval`, объект класса `IdxRangeCatEval` порождает объект класса `IdxRangeCat`, а объект класса `IdxRangeEval` – объект типа `idxrange_t`.

Строковые значения

Строковое значение представлено классом `StringCat`, и используется для хранения текстового представления серии символов в классе `SymDesc`. В объекте класса `StringCat` хранится список объектов, имплементирующих интерфейс `IArray<sym_t>`, обозначаемый как `istring_t`.

Интерфейс `istring_t` имплементируют классы `symrange_t` и `StringValue`. Класс `StringValue` является оберткой для строкового литерала.

Поведение состояний

Класс `StateBlock` описывает блок, определяющий поведение состояний. Он хранит объект класса `IdRefIterEval`, описывающий итерируемую ссылку на состояния, поведение которых должны быть определено, и список объектов класса `Rule`, описывающих отдельные правила переходов.

Класс `Rule` хранит объект класса `IdRefIterEval`, описывающий итерируемую ссылку на символы, к которым применяется правило перехода. Кроме того, в классе `Rule` содержится направление перемещения каретки `Direction`, а также два объекта класса `IdRefEval` – ссылки на записываемый символ и следующее состояние машины.

Класс `IdRefEval` описывает ссылку на символ или состояние, т.е. имя идентификатора и индексы, применяемые к нему. Сами индексы представлены как объекты, имплементирующие интерфейс `IEvaluable`. Это значит, что индексы заданы выражениями, значения которых могут зависеть от некоторых переменных. Метод `eval` класса `IdRefEval` используется для того, чтобы вычислить индексы с учетом текущего контекста, в результате чего получается объект класса `IdRef`, который хранит имя идентификатора и числовые значения индексов при нем.

Аналогично классу `IdRefEval`, класс `IdRefIterEval` описывает итерируемую ссылку на символ или состояние. Индексы итерируемой ссылки либо являются выражениями типа `IEvaluable`, либо объектами класса `IndexIterEval`. Метод `eval` класса `IdRefIterEval` используется для того, чтобы вычислить индексы с учетом текущего контекста, в результате чего получается объект класса `IdRefIter`.

Класс `IndexIterEval` описывает изменение значения одного индекса в итерируемой ссылке. Он хранит последовательность значений индекса в объекте класса `IdxRangeCatEval`.

Объект класса `IdRefIter`, при создании, запускает цикл, описанный итерируемой ссылкой, что означает создание и обновление значений переменных в объекте класса `Context`. Для хранения значений изменяемых индексов используется класс `IndexIter`, объекты которого создаются вызовом метода `eval` класса `IndexIterEval`. В объекте класса `IndexIter` при инициализации создается объект класса `ctx::Variable`, который в свою

очередь добавляет в контекст указанную переменную. Переменная существует, пока существует этот объект.

Арифметические выражения

Класс `IEvaluable` является абстрактным, и описывает метод `inv_sign`, обращающий знак перед выражением, а также метод `eval`, принимающий на вход ссылку на контекст с текущими значениями переменных, и возвращающий на выходе числовое значение. Этот интерфейс имплементируется классами `Number`, `math::Variable` и `Expression`.

Классы `Number` и `math::Variable` являются примитивами, описывающими число и переменную соответственно. Класс `Expression` используется для описания сложных выражений. Он хранит два объекта класса `IEvaluable` и бинарную операцию, применяемую к ним.

3.2. Реализация эмулятора

Эмулятор состоит из двух модулей: программного интерфейса, отвечающего за работу самой Машины Тьюринга, и графического интерфейса, через который пользователь взаимодействует с машиной. Далее приводятся подробности реализации этих модулей.

3.2.1. Программный интерфейс

Программный интерфейс Машины Тьюринга включает в себя класс `Emulator`, который содержит таблицу переходов, текущее состояние машины, данные на ленте и положение каретки.

«Бесконечную» ленту Машины Тьюринга было решено реализовать с помощью связного списка. Таким образом фактическая длина ленты может увеличиваться в любом из направлений настолько, насколько позволяет объем оперативной памяти.

Таблица переходов хранится в виде неупорядоченного ассоциативного списка. В качестве ключа выступает сочетание входного символа и

текущего состояния, а в качестве значения выходной символ, перемещение каретки и следующее состояние. Переход осуществляется путем вызова метода `step`. Если для текущего состояния и считанного символа не задано правило перехода, выбрасывается исключение. Реализация метода `step` приведена в листинге 6.

Листинг 6 – Реализация метода `step` класса `Emulator`

```
void Emulator::step()
{
    if (m_state == STATE_END)
        throw std::logic_error("Final state is already reached");

    Condition cond {.state = m_state, .symbol = *m_tape.car};
    quint64 key = std::bit_cast<quint64>(cond);

    if (!m_table.contains(key)) {
        bool named;
        QString symbol_s = utils::symbolToString(cond.symbol, this->alph(),
&named);

        throw NoRuleError{
            .symbol = named ? symbol_s : QString("'%1'").arg(symbol_s),
            .state = utils::stateToString(cond.state, this->states())
        };
    }

    auto &tr = m_table[key];

    *m_tape.car = tr.symbol;
    m_state = tr.state;

    this->moveCarriage(tr.direction);
}
```

Для того, чтобы загрузить программу в эмулятор, используется метод `loadTable` класса `Loader`. Функция принимает на вход исходный код программы, производит лексический и синтаксический анализ, и загружает программу в объект класса `Emulator`. Реализация функции `loadTable` приведена в листинге 20 приложения Б.

Класс `Loader` также позволяет загружать и выгружать данные из ленты с помощью методов `loadTape` и `readTape`. Реализация этих методов приведена в листинге 7.

Листинг 7 – Реализация методов `loadTape` и `readTape`

```
void Loader::loadTape(QString input, int carPos)
{
    auto input_list = input.toUcs4();
    decltype(m_emu.m_tape.tape) tape(input_list.begin(), input_list.end());

    if (tape.empty())
        tape.push_back(m_emu.symnull());

    m_emu.m_tape.tape = std::move(tape);
    m_emu.m_tape.car = carPos >= 0 ? m_emu.m_tape.tape.begin() :
m_emu.m_tape.tape.end();
    std::advance(m_emu.m_tape.car, carPos);
}

QString Loader::readTape(bool trim) const
{
    if (m_emu.m_tape.tape.empty())
        return QString();

    auto begin = m_emu.m_tape.tape.begin(), end = m_emu.m_tape.tape.end();

    if (trim) {
        while (begin != end && *begin == m_emu.symnull()) ++begin;
        if (begin == end)
            return QString();
        while (end != begin && *(--end) == m_emu.symnull());
        ++end;
    }

    QString output;

    while (begin != end)
        output.append(QString::fromUcs4((char32_t*)&(*begin++), 1));

    return output;
}
```

3.2.2. Графический интерфейс

Интерфейс программы состоит из строки меню, ленты, элементов управления, текстового поля и строки состояния, отображающей имя открытого файла. Внешний вид программы показан на рисунке 4.

Лента, расположенная в верхней части окна программы, реализована так, что количество отображаемых ячеек автоматически подстраивается под размер окна. Ячейка, на которую указывает каретка, выделена, и всегда находится в центре. Каждая ячейка в ленте – это кнопка, по нажатию которой происходит перемещение каретки к выбранной ячейке.

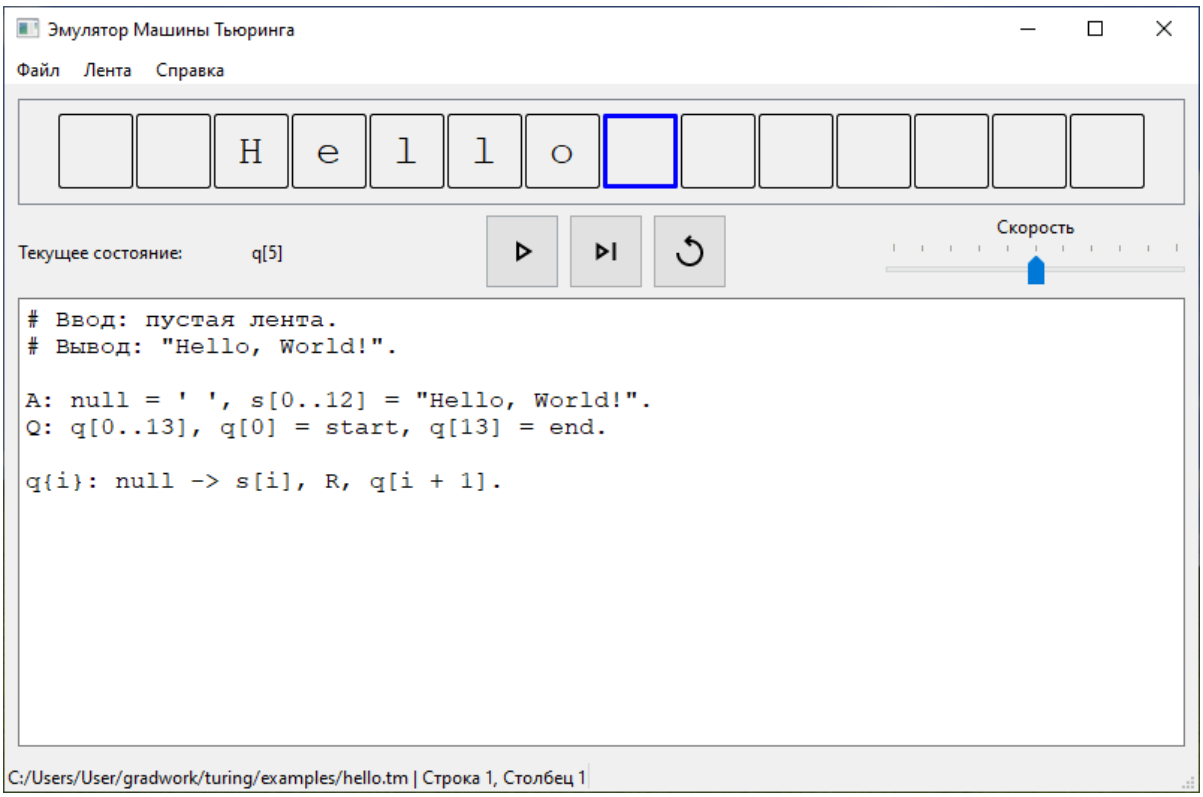

Рисунок 4 – Графический интерфейс программы

Реализация метода `updateCellCount`, вызываемого при изменении размера окна, приведена в листинге 21 приложения Б. В листинге 22 приложения Б приведена реализация функции `updateCellValues`, вызываемой при изменениях данных в ленте.

Панель управления включает в себя следующие элементы управления.

1. Кнопка «запуск». Переводит программу в режим автоматического выполнения программы. Это означает, что машина изменяет состояние с заданной периодичностью. Программа находится в таком режиме до тех пор, пока машина не перейдет в конечное состояние, или пока пользователь не приостановит работу эмулятора повторным нажатием кнопки. Кнопка не доступна, когда машина находится в конечном состоянии.

2. Кнопка «шаг». Выполняет один шаг работы эмулятора. Кнопка недоступна, когда программа находится в режиме автоматического выполнения, а также когда машина находится в конечном состоянии.

3. Кнопка «сброс». Возвращает машину и ленту в изначальное состояние. Если ранее пользователь загружал в ленту данные, они будут заново загружены. Повторное нажатие кнопки «сброса» приведет к очистке ленты. Кнопка недоступна, когда программа находится в режиме автоматического выполнения программы.

4. Ползунок изменения скорости. Регулирует периодичность, с которой машина меняет свое состояние в режиме автоматического выполнения. При крайне правом положении ползунка, программа выполняется моментально, т.е. промежуточные результаты не будут отображаться.

В ходе работы, программа может изменять свой статус, и в зависимости от него, меняется поведение и внешний вид графического интерфейса. Например, статус `READY` означает, что программа готова к запуску эмулятора и, если исходный код был изменен, интерпретатора языка. Статус `RUNNING` означает, что машина запущена в автоматическом режиме, а `PAUSED` значит, что машина начала исполнение программы, но автоматические переходы не выполняются. Наконец, когда машина завершила работу, программа переходит в статус `HALTED`, при котором требуется сбросить состояние машины до начального, перед тем как запустить ее снова.

Выводы по третьей главе

В данной главе были описаны используемые средства реализации, и реализованы все спроектированные компоненты, в частности интерпретатор языка программирования, эмулятор Машины Тьюринга и графический интерфейс эмулятора.

4. ТЕСТИРОВАНИЕ

4.1. Тестирование интерпретатора

Для того, чтобы протестировать интерпретатор разработанного языка программирования, были сформированы исходные коды небольших программ, затрагивающих различные аспекты языка, в том числе содержащие разные ошибки, которые интерпретатор должен обнаружить и сообщить о них. В таблице 2 приведены результаты тестирования. Для каждого теста в таблице 2, в приложении А приводится соответствующий листинг исходного кода программы, номер которого указан в первом столбце таблицы.

Таблица 2 – Тестирование интерпретатора

№	Название теста	Ожидаемый результат	Тест пройден?
1.	Объявление символов	Пустая ячейка обозначается знаком «?». В одну и ту же ячейку в ленте поочередно будут записаны следующие символы, после чего машина завершит работу: `'X', 'A', 'B', 'C', 'D', foo, x[0][2], x[0][1], x[1][2], '!', '?'.`	Да
2.	Объявление состояний	Машина поочередно окажется в следующих состояниях: `start, q[0][1], q[0][2], q[0][3], q[1][0], q[1][1], foo[0], foo[1], foo[2], foo[3], bar, q[1][2], end.`	Да
3.	Итерируемые ссылки и числовые последовательности	В результате работы программы в ленту будет записана следующая последовательность символов: `a[1], a[2], null, a[4], '1', a[6], a[7].` Каретка будет расположена через одну ячейку справа от последнего записанного символа.	Да
4.	Арифметические операции	В результате работы программы, в ленту будет записана следующая последовательность цифр: `8718432661187707.` Каретка будет расположена справа от последнего записанного символа.	Да
5.	Строковые литералы и конкатенация строк	В результате работы программы в ленту будет записана следующая строка: `!"abcdefgh\\hgfedcba"!` Каретка будет расположена справа от последнего записанного символа.	Да

Окончание таблицы 2

№	Название теста	Ожидаемый результат	Тест пройден?
6.	Алфавит не определен	Ошибка: алфавит и состояния должны быть определены в начале.	Да
7.	Пропущен знак точки	Ошибка: ожидался символ точки.	Да
8.	Идентификатор не существует	Ошибка: идентификатор «`foo`» не существует в данном контексте.	Да
9.	Неправильно присвоено значение символу	Ошибка: размер значения не совпадает с размером «`foo`».	Да
10.	Поведение не определено	Ошибка: поведение не определено для состояния `start` и символа `null`.	Да
11.	Индекс выходит за пределы допустимого диапазона	Ошибка: некорректная ссылка.	Да
12.	Ошибка при присвоении значения символу null	Ошибка: невозможно определить значение «`null`» после того, как он был упомянут ранее.	Да
13.	Пропущена кавычка	Ошибка: строковый литерал не был завершен.	Да

4.2. Функциональное тестирование эмулятора

Для проверки корректной работы программы, применялось функциональное тестирование в соответствии с функциональными требованиями, сформулированными во втором разделе. Все тесты пройдены успешно. Результаты функционального тестирования представлены в таблице 3.

Таблица 3 – Функциональное тестирование

№	Название теста	Действия	Тест пройден?
1.	Загрузка таблицы переходов из файла	В меню «Файл» выбрать действие «Открыть»; выбрать файл с программой; убедиться, что содержимое файла отображается в текстовом поле; внести изменения в текстовом поле; в меню «Файл» выбрать действие «Сохранить»; убедиться, что содержимое файла соответствует содержимому текстового поля; в меню «Файл» выбрать действие «Создать»; убедиться, что текстовое поле очищено, а состояние машины и ленты сброшено; внести изменения в текстовое поле; в меню «Файл» выбрать действие «Сохранить»; выбрать путь сохраняемого файла; убедиться, что содержимое файла после сохранения соответствует содержимому текстового поля.	Да

Окончание таблицы 3

№	Название теста	Действия	Тест пройден?
2.	Загрузка данных в ленту из файла	В меню «Лента» выбрать действие «Загрузить ленту»; в диалоговом окне выбрать текстовый файл; убедиться, что в ленте отображаются данные из файла, а каретка указывает на первый символ.	Да
3.	Сохранение данных из ленты в файл	В меню «Лента» выбрать действие «Сохранить ленту»; в диалоговом окне выбрать путь сохраняемого файла; убедиться, что содержимое файла после сохранения соответствует состоянию ленты.	Да
4.	Отображение текущего состояния машины и ленты	В меню «Файл» выбрать действие «Открыть»; выбрать файл с программой; запустить эмулятор нажатием кнопки «Запуск»; убедиться, что лента и отображаемое состояние машины обновляются корректно после каждого шага алгоритма.	Да
5.	Управление	В меню «Файл» выбрать действие «Открыть»; запустить эмулятор нажатием кнопки «Запуск»; приостановить эмулятор после первых нескольких шагов; выполнить несколько шагов нажатием кнопки «Шаг»; изменить скорость выполнения и возобновить работу эмулятора; убедиться, что реакция эмулятора соответствует ожидаемой.	Да
6.	Сброс	В меню «Файл» выбрать действие «Открыть»; в меню «Лента» выбрать действие «Загрузить ленту»; нажать кнопку «Запустить»; после завершения либо приостановки выполнения нажать кнопку «Сброс»; убедиться, что состояние машины – начальное, а лента содержит загруженные ранее данные в первоначальном виде; нажать «Сброс» повторно; убедиться, что лента пуста.	Да
7.	Изменение данных в ленте через диалоговое окно	В меню «Лента» выбрать действие «Изменить ленту»; в диалоговом окне ввести текст; убедиться, что в ленте отображаются введенные данные, а каретка указывает на первый символ.	Да
8.	Ручное перемещение каретки	В меню «Лента» выбрать действие «Изменить ленту»; в диалоговом окне ввести текст; нажать на одну из ячеек ленты, на которую не указывает каретка; убедиться, что каретка переместилась на выбранную ячейку.	Да

Выводы по четвертой главе

В этой главе было проведено тестирование разработанных компонентов. Тестирование интерпретатора и функциональное тестирование эмулятора пройдены успешно, что свидетельствует об их корректной работе.

ЗАКЛЮЧЕНИЕ

В рамках выпускной квалификационной работы были разработаны язык программирования для Машины Тьюринга и эмулятор Машины Тьюринга. Для этого были решены следующие задачи.

1. Выполнен анализ предметной области, изучены соответствующие аналоги и разработаны необходимые требования к системе.
2. Спроектирован язык программирования для Машины Тьюринга и разработан соответствующий интерпретатор.
3. Разработан программный интерфейс для эмуляции работы Машины Тьюринга.
4. Реализован графический интерфейс для эмулятора, позволяющий выполнять программы, написанные на спроектированном языке, и визуализировать работу Машины Тьюринга.
5. Выполнены тестирование разработанного интерпретатора языка программирования, и функциональное тестирование разработанного эмулятора Машины Тьюринга.

Таким образом, поставленные задачи решены, а цель данной выпускной квалификационной работы была достигнута.

В дальнейшем планируется доработка спроектированного языка программирования. Будет добавлена возможность объявлять числовые константы, добавлены новые математические операции и пр.

ЛИТЕРАТУРА

1. Хопкрофт Д., Мотвани Р., Ульман Д. Введение в теорию автоматов, языков и вычислений. // М.: Вильямс, 2002. – 528 с.

2. Herken R. The Universal Turing Machine: A Half-Century Survey. // Springer Verlag, 1995. – 678 с.

3. Brainerd W.S., Landweber L.H. Theory of Computation. // Wiley, 1974. – 336 с.

4. Cook S. The complexity of theorem proving procedures. // Association for Computing Machinery, 1971. – 7 с.

5. Баричев С.Г., Гончаров В.В., Серов Р.Е. Основы современной криптографии. // 3-е изд. – М.: Диалог-МИФИ, 2011. – 176 с.

6. Описание дисциплины «Теория автоматов и алгоритмов» в рамках образовательной программы «Прикладная математика и информатика» Южно-Уральского государственного университета. [Электронный ресурс] URL: https://www.susu.ru/ru/subject/teoriya-avtomatov-i-algoritmov-8 (дата обращения: 01.06.2024 г.).

7. Описание дисциплины «Математическая логика и теория алгоритмов» в рамках образовательной программы «Информационные системы и технологии» Южно-Уральского государственного университета. [Электронный ресурс] URL: https://www.susu.ru/ru/subject/matematicheskaya-logika-i-teoriya-algoritmov-13 (дата обращения: 01.06.2024 г.).

8. Рабочая программа дисциплины «Основы программирования» по направлению подготовки «Фундаментальная информатика и информационные технологии» Челябинского государственного университета. [Электронный ресурс] URL: https://www.csu.ru/faculties/Documents/rpd%5F02.03.02%5Fosn%5Fprogram.pdf (дата обращения: 01.06.2024 г.).

9. Рабочая программа дисциплины «Теория автоматов» по направлению подготовки «Информатика и вычислительная техника» Алтайского государственного университета. [Электронный ресурс] URL: https://www.asu.ru/sveden/education/programs/subject/330576/ (дата обращения: 01.06.2024 г.).

10. Рабочая программа дисциплины «Теория автоматов» по направлению подготовки «Информатика и вычислительная техника» Санкт-Петербургского государственного университета аэрокосмического приборостроения. [Электронный ресурс] URL: https://download.guap.ru/sveden/5248/rpd%5Fteoriya%5Favtomatov%5F357585.pdf (дата обращения: 01.06.2024 г.).

11. Тренажер для изучения универсального исполнителя «Машина Тьюринга» К.Ю. Полякова. [Электронный ресурс] URL: https://kpolyakov.spb.ru/prog/turing.htm (дата обращения: 01.06.2024 г.).

12. Онлайн-симулятор «Turing Machine». [Электронный ресурс] URL: https://turingmachinesimulator.com (дата обращения: 01.06.2024 г.).

13. Официальный сайт Консорциума Юникода. [Электронный ресурс] URL: https://unicode.org (дата обращения: 01.06.2024 г.).

14. Scowen R. Extended BNF – A generic base standard. // IEEE Press, 1993. – 10 с. [Электронный ресурс] URL: https://www.cl.cam.ac.uk/~mgk25/iso-14977-paper.pdf (дата обращения: 01.06.2024 г.).

15. Себеста Р. Основные концепции языков программирования. // Пер. с англ. – 5-е изд. – М.: Вильямс, 2001. – С. 45–52.

16. Буч Г., Рамбо Д., Якобсон И. Язык UML. Руководство пользователя. // 2-е изд. – М.: ДМК Пресс, 2006. – 496 с.

17. Коберн А. Современные методы описания функциональных требований к системам. // Пер. с англ. – М.: Лори, 2012. – 264 с.

18. Онлайн-справочник по языку программирования C++. [Электронный ресурс] URL: https://cplusplus.com/ (дата обращения: 01.06.2024 г.).

19. Официальная документация фреймворка QT. [Электронный ресурс] URL: https://doc.qt.io/ (дата обращения: 01.06.2024 г.).

20. Официальная документация модуля QtWidgets. [Электронный ресурс] URL: https://doc.qt.io/qt-6/qtwidgets-index.html/ (дата обращения: 01.06.2024 г.).

21. Официальное руководство пользователя среды разработки QtCreator. [Электронный ресурс] URL: https://doc.qt.io/qtcreator/ (дата обращения: 01.06.2024 г.).

22. Официальный сайт собрания библиотек C++ Boost. [Электронный ресурс] URL: https://www.boost.org (дата обращения: 01.06.2024 г.).

23. Официальный сайт системы автоматической сборки CMake. [Электронный ресурс] URL: https://cmake.org/ (дата обращения: 01.06.2024 г.).

ПРИЛОЖЕНИЯ

Приложение А. Тестирование интерпретатора

Листинг 1 – Объявление символов

```
A: null = '?', a = 'X', b[0..3] = "ABCD", foo, x[0..1][2..1], x[1][1] =
'!'.
Q: start, end.

start:
      null -> a, N, same;
      a -> b[0], N, same;
      b{i | 0..2} -> b[i + 1], N, same;
      b[3] -> foo, N, same;
      foo -> x[0][2], N, same;
      x[1][1] -> null, N, end;
      x{i}{j} -> x[i + j % 2][1 + j % 2], N, same.
```

Листинг 2 – Объявление состояний

```
A: null.
Q: q[0..1][0..3], q[0][0] = start, q[1][3] = end, foo[0..3], bar.

q[1][1]: null -> same, N, foo[0].
q{i}{j}: null -> same, N, q[i + j / 3][(j + 1) % 4].

foo[3]: null -> same, N, bar.
foo{n}: null -> same, N, foo[n + 1].

bar: null -> same, N, q[1][2].
```

Листинг 3 – Итерируемые ссылки и числовые последовательности

```
A: a[0..7], a[0] = null, one = '1'.
Q: q[0..7], q[0] = start, q[7] = end.

q{n | 0..1 & 6..5 & 3}:
      a{x | 0..n} -> a[x + 1], N, same.

q[4]:
      null -> one, R, q[5].

q{n}:
      a{x} -> same, R, q[n + 1].
```

Листинг 4 – Арифметические операции

```
A: d[0..9] = '0'..'9'.
Q: q[0..16], q[0] = start, q[16] = end.

q{x}:
      null -> d[ -(-12 * x^4 / 7 + 11 * x^3 / -6 - 10 * x^2 / 3 + 9 * x / -2
- 8) % 10 ], R, q[x + 1].
```

Листинг 5 – Строковые литералы и конкатенация строк

```
A: null = ' ', s[0..21] = '!"' & 'abcd' & 'e'..'h' & '\u{5C}\\' & "hg" &
'\u{66}'..'\u{65}' & "d".."a" & "\"!".
Q: q[0..22], q[0] = start, q[22] = end.

q{i}: null -> s[i], R, q[i + 1].
```

Листинг 6 – Алфавит не определен

```
Q: start.

start:
     null -> same, N, end.
```

Листинг 7 – Пропущен знак точки

```
A: null.
Q: start.

start:
     null -> same, N, end
```

Листинг 8 – Идентификатор не существует

```
A: null.
Q: start.

start:
     foo -> same, N, end.
```

Листинг 9 – Неправильно присвоено значение символу

```
A: null, foo = '12'.
Q: start.

start:
     foo -> same, N, end.
```

Листинг 10 – Поведение не определено

```
A: null, foo.
Q: start.

start:
     foo -> same, N, end.
```

Листинг 11 – Индекс выходит за пределы допустимого диапазона

```
A: null, foo[1..3].
Q: start.

start:
      foo{n | 3..4} -> same, N, end.
```

Листинг 12 – Ошибка при присвоении значения символу null

```
A: foo[1..3], foo[1] = null, null = '_'.
Q: start.

start:
      foo{n | 3..4} -> same, N, end.
```

Листинг 13 – Пропущена кавычка

```
A: null, foo = '12.
Q: start.

start:
      foo -> same, N, end.
```

Приложение Б. Реализация различных компонентов

Листинг 14 – Реализация лексического анализатора

```
QList<Token> Tokenizer::tokenize(QString source)
{
    static const QHash<QString, Token::Type> KEYWORDS {
        {"A", Token::KW_A}, {"Q", Token::KW_Q}, {"null", Token::KW_NULL},
{"start", Token::KW_START},
        {"end", Token::KW_END}, {"N", Token::KW_N}, {"L", Token::KW_L},
{"R", Token::KW_R}, {"same", Token::KW_SAME}
    };

    static const QHash<QChar, Token::Type> PUNCTS {
        {':', Token::COLON}, {';', Token::SEMICOLON}, {',', Token::COMMA},
{'.', Token::PERIOD},
        {'=', Token::ASSIGN}, {'&', Token::CAT}, {'_', Token::ANON}, {'|',
Token::ITER},
        {'[', Token::BRACKET_L}, {']', Token::BRACKET_R}, {'{',
Token::BRACE_L}, {'}', Token::BRACE_R},
        {'(', Token::PAR_L}, {')', Token::PAR_R}, {'+', Token::PLUS}, {'-',
Token::MINUS},
        {'*', Token::MUL}, {'/', Token::DIV}, {'%', Token::MOD}, {'^',
Token::POW}
    };

    QList<Token> chain;

    auto it = source.cbegin(), row_begin = source.cbegin(), tok_begin =
source.cbegin();
    SourceRef srcRef{.row = 1};

    bool is_comment = false;

    while (true)
    {
        while (it != source.cend() && (is_comment || it->isSpace())) {
            if (*(it++) == '\n') {
                is_comment = false;
                row_begin = it;
                ++srcRef.row;
            }
        }

        tok_begin = it;
        srcRef.col = tok_begin - row_begin + 1;

        if (it == source.cend())
            break;

        if (*it == '#') {
            is_comment = true;
            continue;
        }

        QStringView tail(&(*it), source.cend() - it);

        Token::Type type = Token::NONE;

        if (tail.startsWith(QLatin1StringView("->"))) {
            type = Token::ARROW;
            it += 2;
```

```
        } else if (tail.startsWith(QLatin1StringView("..."))) {
            type = Token::RANGE;
            it += 2;
        } else if (PUNCTS.contains(*it)) {
            type = PUNCTS.value(*it);
            ++it;
        }

        if (type != Token::NONE) {
            chain.push_back(Token {
                .type = type,
                .srcRef = srcRef
            });
            continue;
        }

        QString name = nextName(tail, it);
        if (!name.isNull()) {
            if (KEYWORDS.contains(name)) {
                chain.push_back(Token{
                    .type = KEYWORDS.value(name),
                    .srcRef = srcRef
                });
            } else {
                chain.push_back(Token{
                    .type = Token::ID,
                    .srcRef = srcRef,
                    .value = name,
                });
            }

            continue;
        }

        Token token = nextLiteral(tail, it, srcRef);
        if (token.type != Token::NONE) {
            token.srcRef = srcRef;
            chain.push_back(token);
            continue;
        }

        throw TokenizerError{ CommonError{
            .srcRef = srcRef,
            .msg = QObject::tr("Unexpected character '%1'").arg(*it)
        }};
    }

    chain.push_back(Token{.type = Token::NONE, .srcRef = srcRef});
    return chain;
}
```

Листинг 15 – Реализация процедуры `nextLiteral`

```
Token Tokenizer::nextLiteral(QStringView tail, QString::const_iterator &it,
SourceRef srcRef)
{
    static QRegularExpression
sym_re(R"raw((["'])|([^"'\n\\])|\\(["'\\])|\\u\{([0-9a-fA-
F]{1,6})\}|.)raw");
```

```
static QRegularExpression num_re(R"raw(^[0-9]+)raw");

Token token;
QStringView view;
bool ok;

auto match = num_re.matchView(tail);
if (match.hasMatch()) {
    view = match.capturedView();

    token.type = Token::NUMBER;
    token.value = view.toLongLong(&ok);

    if (!ok) {
        throw TokenizerError{ CommonError{
            .srcRef = srcRef,
            .msg = QObject::tr("Numeric value is too large")
        }};
    }

    it += view.size();
    return token;
}

QString value;
int skipCounter = 0;

QChar del = tail.first();
if (del != '"' && del != '\'')
    return Token{.type = Token::NONE};

++skipCounter;

auto match_it = sym_re.globalMatchView(tail, 1);
while (match_it.hasNext()) {
    match = match_it.next();

    skipCounter += match.capturedView().size();

    if (match.hasCaptured(1)) {
        view = match.capturedView(1);

        if (view.startsWith(del)) {
            token.type = Token::STRING;
            token.value = value;

            it += skipCounter;
            return token;
        }

        value.append(view);
    }
    else if (match.hasCaptured(2)) {
        view = match.capturedView(2);

        value.append(view);
    }
    else if (match.hasCaptured(3)) {
        view = match.capturedView(3);
```

```
            value.append(view);
        }
        else if (match.hasCaptured(4)) {
            view = match.capturedView(4);
            char32_t sym = view.toUInt(nullptr, 16);

            value.append(QString::fromUcs4(&sym, 1));
        }
        else {
            throw TokenizerError{ CommonError{
                .srcRef = srcRef,
                .msg = QObject::tr("Invalid character '%1' in string
literal").arg(match.capturedView())
            }};
        }
    }

    throw TokenizerError{ CommonError{
        .srcRef = srcRef,
        .msg = QObject::tr("String literal has not been terminated")
    }};
}
```

Листинг 16 – Реализация основных методов класса `IdSpace`

```
void IdSpace::push(const IdDesc &desc, SourceRef srcRef)
{
    if (this->m_nameToId.contains(desc.name))
        throw std::logic_error("name uniqueness must be checked
beforehand");

    id_t id = this->m_stop;
    safe<id_t> stop = id;
    safe<quint64> len = 1;

    try {
        for (const idxrange_t &range : desc.shape)
            len *= range.size();
        stop += len;
    } catch (const std::exception&) {
        throw IdInitError{CommonError{
            .srcRef = srcRef,
            .msg = QObject::tr("Size of '%1' is too big").arg(desc.name)
        }};
    }

    this->m_nameToId.insert(desc.name, id);
    this->m_idToDesc[id] = desc;
    this->m_stop = stop;
}

void IdSpace::setAltId(const IdRef &ref, id_t altId, SourceRef srcRef)
{
    if (!this->isAlt(altId))
        throw std::logic_error("Alternative id must not intersect the id
space");

    id_t id;
    if (!this->get_id_raw(ref, id))
```

```
        throw IdAccessError{CommonError{
            .srcRef = srcRef,
            .msg = QObject::tr("Invalid reference")
        }};
    if (this->m_altId.contains(id)) {
        throw IdInitError{CommonError{
            .srcRef = srcRef,
            .msg = QObject::tr("Cannot re-assign value")
        }};
    }
    this->m_altId.insert(id, altId);
}

id_t IdSpace::getId(const IdRef &ref, SourceRef srcRef) const
{
    id_t id;
    if (!this->get_id_raw(ref, id)) {
        throw IdAccessError{CommonError{
            .srcRef = srcRef,
            .msg = QObject::tr("Invalid reference")
        }};
    }
    if (!this->m_altId.contains(id))
        return id;
    return this->m_altId.value(id);
}

IdRef IdSpace::getRef(id_t id) const
{
    if (id < this->m_start || id >= this->m_stop)
        throw std::logic_error("Id is not in the id space");

    auto it = --this->m_idToDesc.upper_bound(id);
    id_t base_id = it->first;
    const IdDesc &desc = it->second;

    IdRef ref;
    ref.name = desc.name;

    quint64 pos = id - base_id;
    for (auto it = desc.shape.crbegin(); it != desc.shape.crend(); ++it) {
        ref.idx.push_front((*it)[pos % it->size()]);
        pos /= it->size();
    }

    return ref;
}

IdDesc IdSpace::getDesc(name_t name, SourceRef srcRef) const
{
    if (!this->m_nameToId.contains(name))
        throw IdAccessError{CommonError{
            .srcRef = srcRef,
            .msg = QObject::tr("Identifier '%1' does not exist in this
context").arg(name)
        }};
    auto id = this->m_nameToId.value(name);

    return this->m_idToDesc.at(id);
}
```

Листинг 17 – Реализация основных методов класса `SymSpace`

```
void SymSpace::insert(SymDesc &&desc, SourceRef srcRef)
{
    if (this->m_nameToDesc.contains(desc.name))
        throw std::logic_error("name uniqueness must be checked
beforehand");

    safe<quint64> len = 1;
    try {
        for (const idxrange_t &range : desc.shape)
            len *= range.size();
    } catch (const std::exception&) {
        throw IdInitError{CommonError{
            .srcRef = srcRef,
            .msg = QObject::tr("Size of '%1' is too big").arg(desc.name)
        }};
    }

    if (desc.value.size() != len)
        throw IdInitError{CommonError{
            .srcRef = srcRef,
            .msg = QObject::tr("Value size doesn't match size of
'%1'").arg(desc.name)
        }};
    this->m_nameToDesc[desc.name] = std::move(desc);
}

sym_t SymSpace::getSym(const IdRef &ref, SourceRef srcRef) const
{
    if (!this->m_nameToDesc.contains(ref.name)) {
        throw IdAccessError{CommonError{
            .srcRef = srcRef,
            .msg = QObject::tr("Identifier '%1' does not exist in this
context").arg(ref.name)
        }};
    }

    const SymDesc &desc = this->m_nameToDesc.at(ref.name);
    uint64_t pos;
    if (!idxToPos(ref.idx, desc.shape, pos)) {
        throw IdAccessError{CommonError{
            .srcRef = srcRef,
            .msg = QObject::tr("Invalid reference")
        }};
    }
    return desc.value[pos];
}

const SymDesc& SymSpace::getDesc(name_t name, SourceRef srcRef) const
{
    if (!this->m_nameToDesc.contains(name)) {
        throw IdAccessError{CommonError{
            .srcRef = srcRef,
            .msg = QObject::tr("Identifier '%1' does not exist in this
context").arg(name)
        }};
    }
    return this->m_nameToDesc.at(name);
}
```

Листинг 18 – Реализация синтаксического анализа объявлений состояний

```
States::States(QList<Token>::const_iterator begin,
QList<Token>::const_iterator end, ctx::Context &context)
    : context(context)
    , states(2)
{
    if (begin == end) {
        throw ParseError{ CommonError{
            .srcRef = begin->srcRef,
            .msg = QObject::tr("Expected state declaration")
        }};
    }

    auto it = begin;
    while (this->addNextDeclaration(it, end));
}

bool States::addNextDeclaration(QList<Token>::const_iterator &it,
QList<Token>::const_iterator end)
{
    if (it == end)
        return false;

    id::IdRef ref;
    id::IdDesc desc;
    enum {NONE, KW, DESC, REF} lval_type = NONE;

    auto it_lvalue = it;
    if (it->type == Token::KW_START || it->type == Token::KW_END) {
        ++it;
        lval_type = KW;
    }
    else if (it->type == Token::ID) {
        id::name_t name = it->value.toString();
        ++it;
        auto idx_or_shape = getIdxOrShape(&this->context, it, end, ref.idx,
&desc.shape);
        if (idx_or_shape == IdxOrShape_e::IDX) {
            lval_type = REF;
            ref.name = name;
        }
        else {
            if (this->context.has(name)) {
                throw ctx::NameOccupiedError{ CommonError{
                    .srcRef = it_lvalue->srcRef,
                    .msg = QObject::tr("Name '%1' is already
occupied").arg(name)
                }};
            }
            this->context.other_names.insert(name);
            lval_type = DESC;
            desc.name = name;
        }
    }
    else {
        throw ParseError{ CommonError{
            .srcRef = it->srcRef,
            .msg = QObject::tr("Expected identifier, 'start' or 'end'")
        }};
```

```
    }

    if (it == end || it->type == Token::COMMA) {
        if (lval_type == REF) {
            throw ParseError{ CommonError{
                .srcRef = it->srcRef,
                .msg = QObject::tr("Expected '='")
            }};
        }
        if (lval_type == DESC)
            this->states.push(desc, it_lvalue->srcRef);
    }
    else if (it->type == Token::ASSIGN) {
        if (lval_type != REF) {
            throw ParseError{ CommonError{
                .srcRef = it->srcRef,
                .msg = QObject::tr("Unexpected '='")
            }};
        }
        ++it;

        if (it == end) {
            throw ParseError{ CommonError{
                .srcRef = it->srcRef,
                .msg = QObject::tr("Expected value to assign")
            }};
        }
        switch (it->type) {
        case Token::KW_START:
            this->states.setAltId(ref, emu::STATE_START, it_lvalue-
>srcRef);
            break;
        case Token::KW_END:
            this->states.setAltId(ref, emu::STATE_END, it_lvalue->srcRef);
            break;
        default:
            throw ParseError{ CommonError{
                .srcRef = it->srcRef,
                .msg = QObject::tr("Expected 'start' or 'end'")
            }};
        }
        ++it;
    }
    else {
        throw ParseError{ CommonError{
            .srcRef = it->srcRef,
            .msg = QObject::tr("Unexpected token")
        }};
    }
    if (it != end) {
        if (it->type != Token::COMMA) {
            throw ParseError{ CommonError{
                .srcRef = it->srcRef,
                .msg = QObject::tr("Unexpected token")
            }};
        }
        ++it;
    }
    return true;
}
```

Листинг 19 – Реализация синтаксического анализа объявлений символов

```
Alphabet::Alphabet(QList<Token>::const_iterator begin,
QList<Token>::const_iterator end, ctx::Context &context)
    : context(context)
    , alph(0x80'00'00'00)
    , null_value(0)
    , is_null_declared(false)
    , is_null_requested(false)
{
    if (begin == end) {
        throw ParseError{ CommonError{
            .srcRef = begin->srcRef,
            .msg = QObject::tr("Expected symbol declaration")
        }};
    }

    auto it = begin;
    while (this->addNextDeclaration(it, end));
}

bool Alphabet::addNextDeclaration(QList<Token>::const_iterator &it,
QList<Token>::const_iterator end)
{
    if (it == end)
        return false;

    id::IdRef ref;
    id::SymDesc desc;

    enum {NONE, KW_NULL, DESC, REF} lval_type = NONE;

    auto it_lvalue = it;

    if (it->type == Token::KW_NULL) {
        if (this->is_null_declared || this->is_null_requested) {
            throw ParseError{ CommonError{
                .srcRef = it->srcRef,
                .msg = QObject::tr("Cannot declare 'null' after it has been
mentioned before")
            }};
        }

        ++it;

        this->is_null_declared = true;
        lval_type = KW_NULL;
    }
    else if (it->type == Token::ID) {
        id::name_t name = it->value.toString();
        ++it;

        auto idx_or_shape = getIdxOrShape(&this->context, it, end, ref.idx,
&desc.shape);

        if (idx_or_shape == IdxOrShape_e::IDX) {
            lval_type = REF;
            ref.name = name;
        }
        else {
```

```
            if (this->context.has(name)) {
                throw ctx::NameOccupiedError{ CommonError{
                    .srcRef = it_lvalue->srcRef,
                    .msg = QObject::tr("Name '%1' is already
occupied").arg(name)
                }};
            }
            this->context.other_names.insert(name);

            lval_type = DESC;
            desc.name = name;
        }
    }
    else {
        throw ParseError{ CommonError{
            .srcRef = it->srcRef,
            .msg = QObject::tr("Expected identifier or 'null'")
        }};
    }

    if (it == end || it->type == Token::COMMA) {
        if (lval_type == REF) {
            throw ParseError{ CommonError{
                .srcRef = it->srcRef,
                .msg = QObject::tr("Expected '='")
            }};
        }

        if (lval_type == DESC)
            this->alph.push(desc, it_lvalue->srcRef);
    }
    else if (it->type == Token::ASSIGN) {
        ++it;

        if (it == end) {
            throw ParseError{ CommonError{
                .srcRef = it->srcRef,
                .msg = QObject::tr("Expected value to assign")
            }};
        }

        if (lval_type == DESC) {
            auto rval_lbound = it;
            auto rval_rbound = it = nextToken(it, end, Token::COMMA);
            desc.value.parse(rval_lbound, rval_rbound);

            this->alph_sym.insert(std::move(desc), it_lvalue->srcRef);
        } else {
            id::id_t value;

            if (it->type == Token::STRING) {
                auto value_s = it->value.toString().toUcs4();

                if (value_s.size() != 1) {
                    throw ParseError{ CommonError{
                        .srcRef = it->srcRef,
                        .msg = QObject::tr("String literal must be of
length 1")
                    }};
                }
```

```
                ++it;
                value = value_s.front();
            }
            else if (it->type == Token::KW_NULL && lval_type == REF) {
                ++it;
                value = this->null_value;
                this->is_null_requested = true;
            }
            else {
                throw ParseError{ CommonError{
                    .srcRef = it->srcRef,
                    .msg = QObject::tr("Unexpected token")
                }};
            }

            if (lval_type == KW_NULL) {
                this->null_value = value;
            } else {
                this->alph.setAltId(ref, value, it_lvalue->srcRef);
            }
        }
    }
    else {
        throw ParseError{ CommonError{
            .srcRef = it->srcRef,
            .msg = QObject::tr("Unexpected token")
        }};
    }

    if (it != end) {
        if (it->type != Token::COMMA) {
            throw ParseError{ CommonError{
                .srcRef = it->srcRef,
                .msg = QObject::tr("Unexpected token")
            }};
        }
        ++it;
    }

    return true;
}
```

Листинг 20 – Реализация функции `loadTable`

```
void Loader::loadTable(QString source, bool preserveTape)
{
    auto chain = Tokenizer::tokenize(source);

    Parser parser(chain.cbegin(), chain.cend() - 1);
    Emulator emu(parser.alph, parser.states);

    emu::Condition cond;
    emu::Transition tr;

    for (auto &block : parser.blocks) {
        id::IdRefIter state_iter;
        id::IdRef state_ref;
        id::id_t state_id;
```

```
        switch (block.refiter_type) {
        case parser::ITER:
            state_ref.name = block.refiter.getName();

            state_iter = block.refiter.eval(
                parser.context,
                parser.states->states.getDesc(state_ref.name,
block.refiter.getSrcRef()).shape
            );

            state_iter.value(state_ref.idx);

            state_id = parser.states->states.getId(state_ref,
block.refiter.getSrcRef());
            break;

        case parser::KW_START:
            state_id = emu::STATE_START;
            break;

        default:
            throw std::logic_error("");
        }

        do {
            if (state_id == emu::STATE_END) {
                continue;
            }

            for (auto &rule : block.rules) {
                id::IdRefIter sym_iter;
                id::IdRef sym_ref;
                id::id_t sym_id;

                switch (rule.refiter_type) {
                case parser::ITER:
                    sym_ref.name = rule.refiter.getName();

                    sym_iter = rule.refiter.eval(
                        parser.context,
                        parser.alph->getIdDesc(sym_ref.name,
rule.refiter.getSrcRef()).shape
                    );

                    sym_iter.value(sym_ref.idx);

                    sym_id = parser.alph->getId(sym_ref,
rule.refiter.getSrcRef());
                    break;

                case parser::KW_NULL:
                    sym_id = emu.symnull();
                    break;

                default:
                    throw std::logic_error("");
                }

                do {
```

```
                    emu::Condition cond {.state = state_id, .symbol =
sym_id};

                    if (emu.getRule(cond) != nullptr)
                        continue;

                    tur::emu::Transition tr {.direction = rule.dir};

                    switch (rule.symbol_type) {
                    case parser::REF:
                        tr.symbol = parser.alph-
>getId(rule.symbol.eval(&parser.context), rule.symbol.getSrcRef());
                        break;
                    case parser::KW_SAME:
                        tr.symbol = sym_id;
                        break;
                    case parser::KW_NULL:
                        tr.symbol = emu.symnull();
                        break;
                    default:
                        throw std::logic_error("");
                    }

                    switch (rule.state_type) {
                    case parser::REF:
                        tr.state = parser.states-
>states.getId(rule.state.eval(&parser.context), rule.state.getSrcRef());
                        break;
                    case parser::KW_SAME:
                        tr.state = state_id;
                        break;
                    case parser::KW_START:
                        tr.state = tur::emu::STATE_START;
                        break;
                    case parser::KW_END:
                        tr.state = tur::emu::STATE_END;
                        break;
                    default:
                        throw std::logic_error("");
                    }

                    emu.addRule(cond, tr);
                } while (
                    sym_iter.next()
                    && sym_iter.value(sym_ref.idx)
                    && (sym_id = parser.alph->getId(sym_ref,
rule.refiter.getSrcRef()), true)
                );
            }
        } while (
            state_iter.next()
            && state_iter.value(state_ref.idx)
            && (state_id = parser.states->states.getId(state_ref,
block.refiter.getSrcRef()), true)
        );
    }

    if (preserveTape) {
        auto null_old = this->m_emu.symnull(), null_new = emu.symnull();
```

```
        emu.m_tape = std::move(this->m_emu.m_tape);
        for (auto &val : emu.m_tape.tape) {
            if (val == null_old)
                val = null_new;
        }
    }

    this->m_emu = std::move(emu);
}
```

Листинг 21 – Реализация функции `updateCellCount`

```
void MainWindow::updateCellCount()
{
    int left, right;
    ui->gridLayout->getContentsMargins(&left, nullptr, &right, nullptr);
    int tapeWidth = size().width() - (left + right);
    int hspacing = ui->tape->spacing();
    int count = ui->tape->count(), newCount = (tapeWidth + hspacing - 20) /
(Cell::CELL_SIZE + hspacing);

    while (count < newCount) {
        auto *cell = new Cell(this);
        QObject::connect(cell, &QPushButton::clicked, [this, cell](){
            this->emu.moveCarriage(cell->diff);
            this->updateCellValues();
        });
        ui->tape->addWidget(cell);
        ++count;
    }

    for (int index = count - 1; index >= newCount; --index) {
        auto *item = ui->tape->itemAt(index);
        ui->tape->removeItem(item);
        delete item->widget();
        delete item;
    }

    updateCellValues();
}
```

Листинг 22 – Реализация функции `updateCellValues`

```
void MainWindow::updateCellValues()
{
    auto &tape = emu.tape();
    int index, diff;
    int cellsCount = ui->tape->count();
    int carCellIndex = cellsCount / 2;

    auto symnull = emu.symnull();

    for (index = 0; index < cellsCount; ++index) {
        auto *cell = dynamic_cast<Cell*>(ui->tape->itemAt(index)-
>widget());
        cell->setValue(symnull, this->emu.alph());
        cell->setSelected(false);
    }
```

```
    auto car = emu.carriage(), it = car;

    for (it = car, index = carCellIndex, diff = 0; ++it != tape.end() &&
++index < cellsCount; ) {
        auto *cell = dynamic_cast<Cell*>(ui->tape->itemAt(index)-
>widget());
        cell->setValue(*it, this->emu.alph());
    }

    dynamic_cast<Cell*>(ui->tape->itemAt(carCellIndex)->widget())-
>setSelected(true);

    for (it = car, index = carCellIndex, diff = 0; index >= 0; --it, --
index) {
        auto *cell = dynamic_cast<Cell*>(ui->tape->itemAt(index)-
>widget());
        cell->setValue(*it, this->emu.alph());

        if (it == tape.begin())
            break;
    }

    for (index = 0, diff = -carCellIndex; index < cellsCount; ++index,
++diff){
        auto *cell = dynamic_cast<Cell*>(ui->tape->itemAt(index)-
>widget());
        cell->diff = diff;
    }
}
```

Приложение В. Обзор аналогов

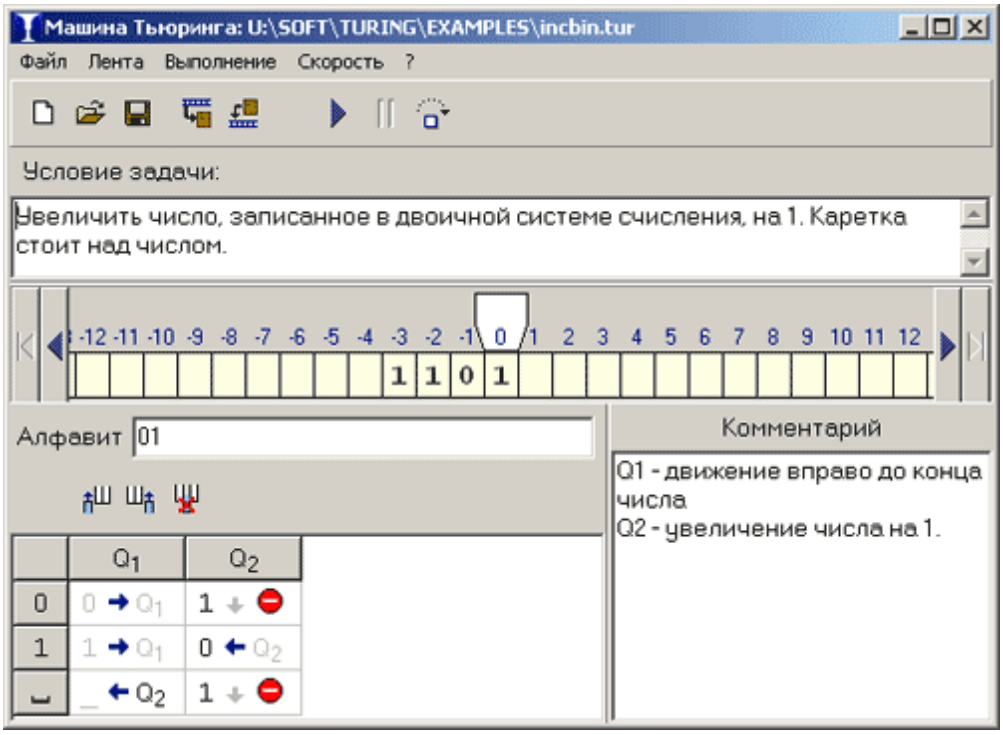

Рисунок 1 – Интерфейс программы «Машина Тьюринга» [11]

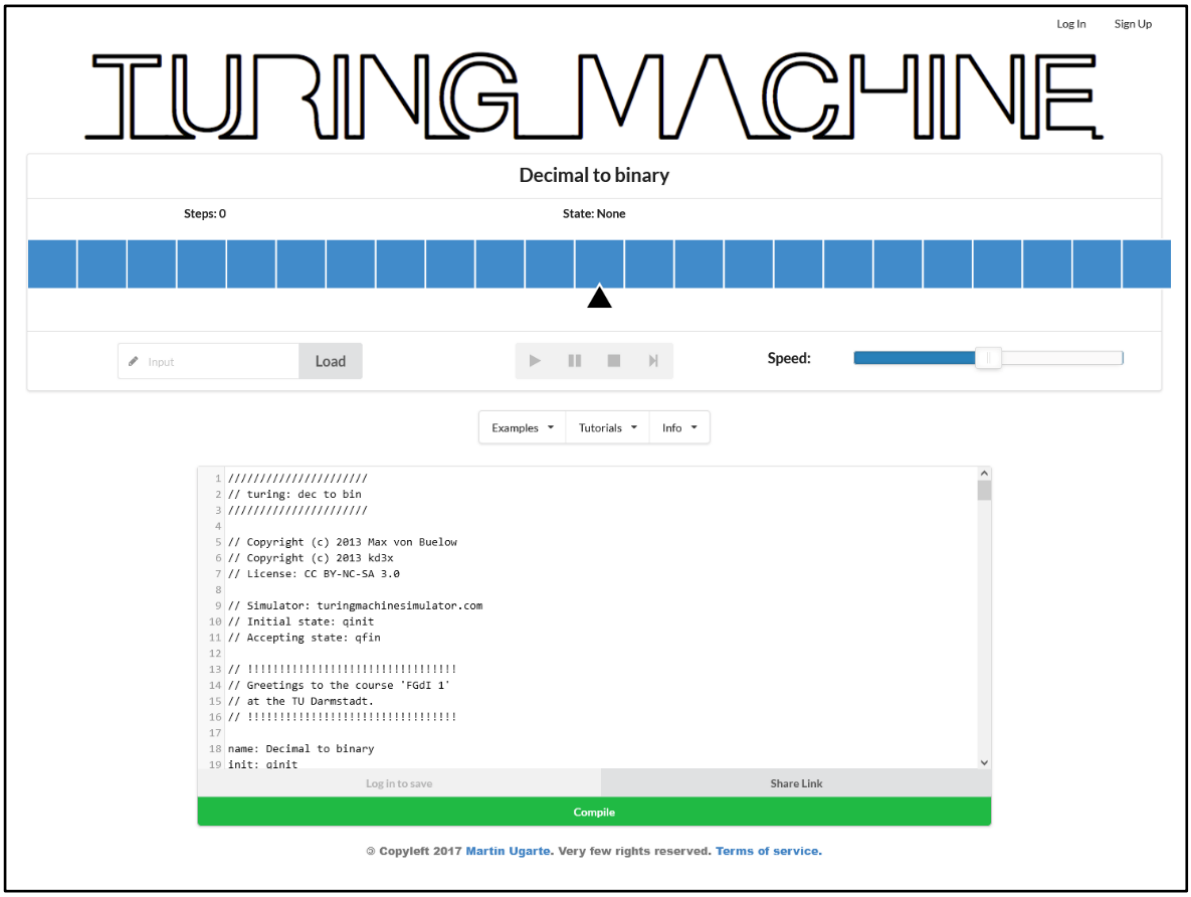

Рисунок 2 – Интерфейс веб-приложения «Turing Machine» [12]

Приложение Г. Диаграммы

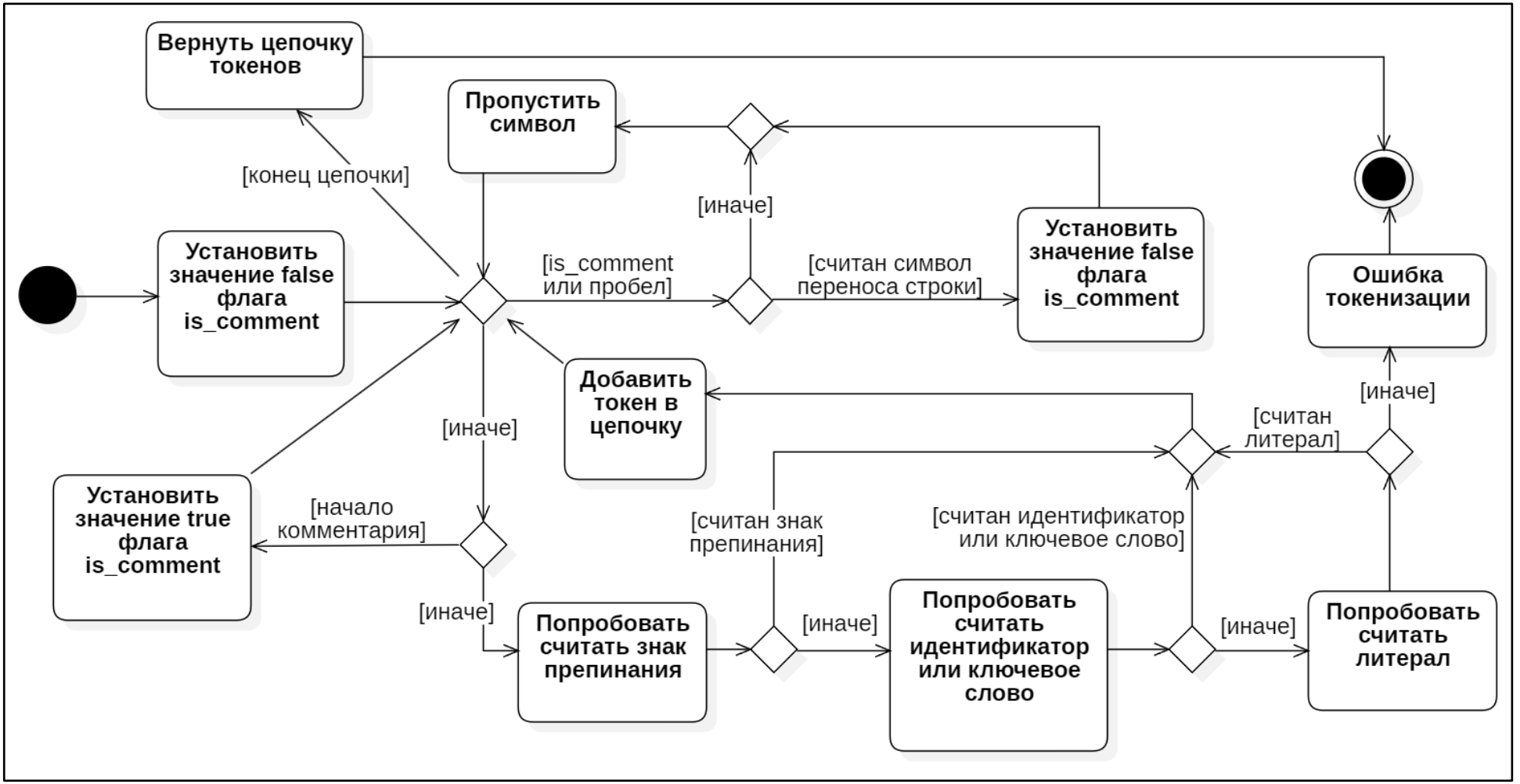

Рисунок 3 – Диаграмма деятельности лексического анализатора

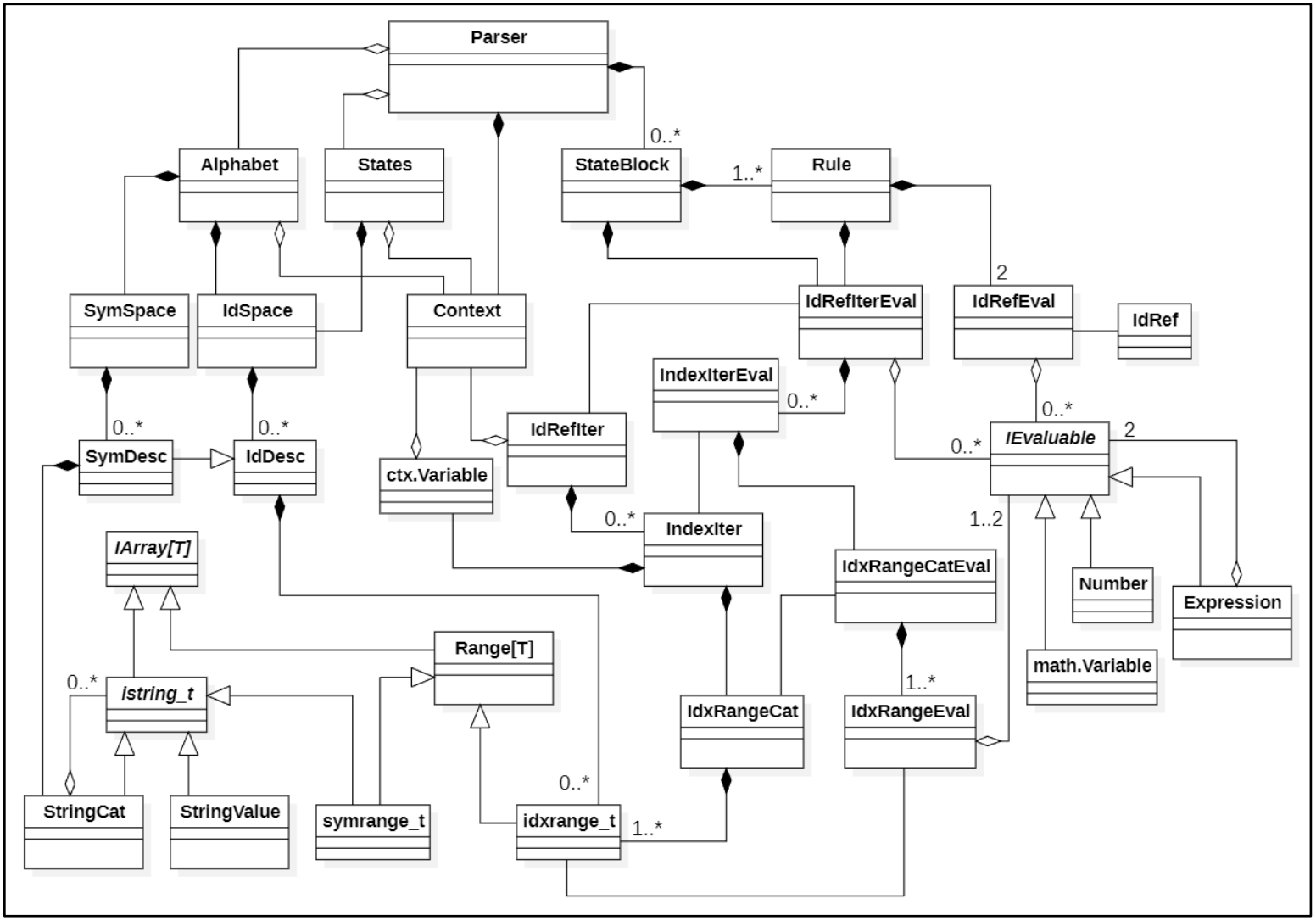

Рисунок 4 – Диаграмма классов синтаксического анализатора