**Содержание**

[Введение 5](#_Toc154054381)

[1. Спецификация языка программирования 6](#_Toc154054382)

[1.1 Характеристика языка программирования 6](#_Toc154054383)

[1.2 Определение алфавита языка программирования 6](#_Toc154054384)

[1.3 Применяемые сепараторы 7](#_Toc154054385)

[1.4 Применяемые кодировки 7](#_Toc154054386)

[1.5 Типы данных 7](#_Toc154054387)

[1.6 Преобразование типов данных 10](#_Toc154054388)

[1.7 Идентификаторы 10](#_Toc154054389)

[1.8 Литералы 11](#_Toc154054390)

[1.9 Объявление данных 11](#_Toc154054391)

[1.10 Инициализация данных 12](#_Toc154054392)

[1.11 Инструкции языка 12](#_Toc154054393)

[1.12 Операции языка 13](#_Toc154054394)

[1.13 Выражения и их вычисления 14](#_Toc154054395)

[1.14 Конструкции языка 14](#_Toc154054396)

[1.15 Область видимости идентификаторов 14](#_Toc154054397)

[1.16 Семантические проверки 15](#_Toc154054398)

[1.17 Распределение оперативной памяти на этапе выполнения 15](#_Toc154054399)

[1.18 Стандартная библиотека и ее состав 15](#_Toc154054400)

[1.20 Точка входа 17](#_Toc154054401)

[1.21 Препроцессор 17](#_Toc154054402)

[1.22 Соглашения о вызове 17](#_Toc154054403)

[1.23 Объектный код 17](#_Toc154054404)

[1.24 Классификация сообщений транслятора 18](#_Toc154054405)

[1.25 Контрольный пример 18](#_Toc154054406)

[2. Структура транслятора 19](#_Toc154054407)

[2.1 Компоненты транслятора, их назначение и принципы взаимодействия 19](#_Toc154054408)

[2.2 Перечень входных параметров транслятора 20](#_Toc154054409)

[2.3 Протоколы, формируемые транслятором и их содержимое 20](#_Toc154054410)

[3 Разработка лексического анализатора 22](#_Toc154054411)

[3.1 Структура лексического анализатора 22](#_Toc154054412)

[3.2 Входные и выходные данные лексического анализатор 23](#_Toc154054413)

[3.3 Параметры лексического анализатора 23](#_Toc154054414)

[3.4 Алгоритм лексического анализа 23](#_Toc154054415)

[3.5 Контроль входных символов 24](#_Toc154054416)

[3.6 Удаление избыточных символов 24](#_Toc154054417)

[3.7 Перечень ключевых слов 25](#_Toc154054418)

[3.8 Основные структуры данных 26](#_Toc154054419)

[3.9 Структура и перечень сообщений лексического анализатора 28](#_Toc154054420)

[3.10 Принцип обработки ошибок 28](#_Toc154054421)

[3.11 Контрольный пример 29](#_Toc154054422)

[4. Разработка синтаксического анализатора 30](#_Toc154054423)

[4.1 Структура синтаксического анализатора 30](#_Toc154054424)

[4.2 Контекстно-свободная грамматика, описывающая синтаксис языка 30](#_Toc154054425)

[4.3 Построение конечного магазинного автомата 32](#_Toc154054426)

[4.4 Основные структуры данных 33](#_Toc154054427)

[4.5 Описание алгоритма синтаксического разбора 33](#_Toc154054428)

[4.6 Параметры синтаксического анализатора и режимы его работы 34](#_Toc154054429)

[4.7 Структура и перечень сообщений синтаксического анализатора 34](#_Toc154054430)

[4.8 Принцип обработки ошибок 35](#_Toc154054431)

[4.9 Контрольный пример 35](#_Toc154054432)

[5 Разработка семантического анализатора 36](#_Toc154054433)

[5.1 Структура семантического анализатора 36](#_Toc154054434)

[5.2 Функции семантического анализатора 36](#_Toc154054435)

[5.3 Структура и перечень сообщений семантического анализатора 36](#_Toc154054436)

[5.4 Принцип обработки ошибок 37](#_Toc154054437)

[5.5 Контрольный пример 37](#_Toc154054438)

[6. Вычисление выражений 39](#_Toc154054439)

[6.1 Выражения, допускаемые языком 39](#_Toc154054440)

[6.2 Польская запись и принцип её построения 39](#_Toc154054441)

[6.3 Программная реализация обработки выражений 39](#_Toc154054442)

[6.4 Контрольный пример 39](#_Toc154054443)

[7. Генерация кода 40](#_Toc154054444)

[7.1 Структура генератора кода 40](#_Toc154054445)

[7.2 Представление типов данных в оперативной памяти 40](#_Toc154054446)

[7.3 Статическая библиотека 40](#_Toc154054447)

[7.4 Особенности алгоритма генерации кода 40](#_Toc154054448)

[7.5 Параметры, управляющие генерацией кода 41](#_Toc154054449)

[7.6 Контрольный пример 41](#_Toc154054450)

[8. Тестирование транслятора 42](#_Toc154054451)

[8.1 Общие положения 42](#_Toc154054452)

[8.2 Результаты тестирования 42](#_Toc154054453)

[Заключение 44](#_Toc154054454)

[Список использованных источников 45](#_Toc154054455)

[Приложение А 46](#_Toc154054456)

[Приложение Б 47](#_Toc154054457)

[Приложение В 51](#_Toc154054458)

[Приложение Г 61](#_Toc154054459)

[Приложение Д 62](#_Toc154054460)

# Введение

Цель курсового проекта - разработка компилятора для нового языка программирования, VKS-2023, ориентированного на выполнение логических операций с числами и строками. Компилятор - это специализированный инструмент, который преобразует код, написанный на одном языке программирования, в этом случае VKS-2023, в код на другом языке, как JavaScript.

Процесс компиляции разделяется на два основных этапа: анализ и генерацию кода. Анализ включает в себя разбор исходной программы на составные части и формирование ее промежуточного представления. Генерация заключается в создании итоговой программы на JavaScript, основываясь на промежуточном представлении.

Компилятор VKS-2023 состоит из следующих составных частей:

* лексический анализатор;
* синтаксический анализатор;
* семантический анализатор;
* генератор исходного кода на языке JavaScript.

Исходя из цели курсового проекта, были определены следующие задачи:

* разработка спецификации языка программирования;
* разработка структуры транслятора;
* разработка лексического анализатора;
* разработка синтаксического анализатора;
* преобразование выражений;
* генерация кода на язык JavaScript;
* тестирование транслятора.

В каждой главе курсового проекта будут представлены решения, соответствующие установленным задачам.

# 1. Спецификация языка программирования

## **1.1 Характеристика языка программирования**

VKS-2023 является компилируемым процедурным строго типизированным языком.

Поддерживает 5 типов данных: строковый (str) и целочисленные (number и byte) , логический(bool), html объект (htmlobj). В стандартной библиотеке языка содержится 16 функций: для работы с числами – квадратный корень из числа sqrt(number x), возведение числа в произвольную степень pow(number x, number degree), 3 функции для работы со строками – получение длинны строки strsize(str string), объединение строк concat(str string1, str string2), перевод числа в строку itos(number x), функция вывода в консоль браузера – console(str string).

**1.2 Определение алфавита языка программирования**

В основе алфавита VKS-2023 лежит таблица символов Windows-1251, которая представлена на рисунке 1.1.



Рисунок 1.1 – Таблица кодировки Windows-1251

Разрешены латинские и кириллические символы, символы-сепараторы, символы операций и другие.

Алфавит языка VKS-2023 состоит из следующих множеств символов:

Для записи инструкций языка используются символы: [a…z], [A…Z].

Для записи литералов (строковых и целочисленных) используют символы: [a…я], [А…Я], [a…z], [A…Z], [0…9].

В качестве сепараторов и специальных символов используются: [ ] ( ) , ; : # + - / \* > < & !. “пробел”

**1.3 Применяемые сепараторы**

Сепараторы необходимы для разделения операций языка. Сепараторы, используемые в языке программирования VKS-2023, приведены в таблице 1.1.

Таблица 1.1 – Применяемые сепараторы

|  |  |
| --- | --- |
| Символы | Назначение |
| **{** … **}** | Блок функции или условной конструкции/цикла |
| **(** … **)** | Блок фактических или формальных параметров функции, а также приоритет арифметических операций |
| **,** | Разделитель параметров функций |
| **+ - \*** | Арифметические операции |
| **> < => <=**  **== !=** | Логические операции (операции сравнения: больше, меньше, больше или равно, меньше или равно, проверка на равенство, неравенство), используемые в условии цикла/условной конструкции. |
| **;** | Разделитель программных конструкций |
| ‘пробел’ | Разделитель цепочек. Допускается везде кроме названий идентификаторов и ключевых слов |

Таким образом язык программирования VKS-2023 использует все приведённые выше символы.

**1.4 Применяемые кодировки**

При трансляции исходного кода применяется кодировка Windows-1251. Описание кодировки представлено в пункте 1.2

## **1.5 Типы данных**

Тип данных — это характеристика значений, которые принимает некоторая переменная или выражение. Есть семь типов данных: логические, целочисленные, с плавающей запятой, строковые, указатели, идентификационные, абстрактные

В языке VKS-2023 реализованы 3 фундаментальных типа данных: знаковый целочисленный (number и byte), строковый и логические, которые описываются в таблице 1.2. Также реализован специальный тип данных htmlobj. Пользовательские типы данных не поддерживаются.

Реализация данных типов позволяет организовать работу множества функций для работы как с числами, так и со строками, однако язык VKS-2023 не предусматривает перевод из одного типа в другой

Таблица 1.2 – Типы данных языка VKS-2023

|  |  |
| --- | --- |
| Тип данных | Характеристика |
| Строковый тип данных str | Фундаментальный тип данных, используемый для объявления строк. Без явно указанной инициализации переменной, присваивается нулевое значение (пустая строка).  Максимальное количество символов – 255. |
| Целочисленный знаковый тип данных number | Фундаментальный тип данных, используемый для объявления целочисленных данных. Без явно указанной инициализации переменной, присваивается нулевое значение. Представляет только положительное целое число.  Максимальное значение: 2147483647.  Минимальное значение: -2147483647.  **+** (бинарный) – оператор сложения;  - (бинарный) – оператор вычитания;  **\*** (бинарный) – оператор умножения;  **=** (бинарный) – оператор присваивания  В качестве участников булевого выражения оператора поддерживаются следующие логические операции:  **>** (бинарный) – оператор «больше»;  **<** (бинарный) – оператор «меньше»;  **=>** (бинарный) – оператор «больше или равно»;  **<=**(бинарный) – оператор «меньше или равно»;  **==** (бинарный) – оператор проверки на равенство.  **!=** (бинарный) – оператор проверки на неравенство. |

Продолжение таблицы 1.2

|  |  |
| --- | --- |
| Тип данных | Характеристика |
| Целочисленный знаковый тип данных byte | Фундаментальный тип данных, используемый для объявления целочисленных данных. Без явно указанной инициализации переменной, присваивается нулевое значение. Представляет только положительное целое число.  Максимальное значение: 127.  Минимальное значение: -128.  **+** (бинарный) – оператор сложения;  - (бинарный) – оператор вычитания;  **\*** (бинарный) – оператор умножения;  **=** (бинарный) – оператор присваивания  В качестве участников булевого выражения оператора поддерживаются следующие логические операции:  **>** (бинарный) – оператор «больше»;  **<** (бинарный) – оператор «меньше»;  **=>** (бинарный) – оператор «больше или равно»;  **<=**(бинарный) – оператор «меньше или равно»;  **==** (бинарный) – оператор проверки на равенство.  **!=** (бинарный) – оператор проверки на неравенство. |
| Логический тип данных bool | Фундаментальный тип данных, используемый для записи логический значений. Без явно указанной инициализации переменной, присваивается false.  Принимает значения true или false.  Поддерживает значения:  **>** (бинарный) – оператор «больше»;  **<** (бинарный) – оператор «меньше»;  **=>** (бинарный) – оператор «больше или равно»;  **<=**(бинарный) – оператор «меньше или равно»; |

Окончание таблицы 1.2

|  |  |
| --- | --- |
| Тип данных | Характеристика |
|  | **==** (бинарный) – оператор проверки на равенство.  **!=** (бинарный) – оператор проверки на неравенство. |
| Специальный тип данных HTMLOBJ | Тип данных, используемый для хранения результатов работы функций с html.  Задать значения можно только с помощью функций. |

В таблице представлены четыре типа данных: строковый (str) и целочисленный (number и byte), логический (bool), специальный (HTMLOBJ). Задание значения для целочисленных данным возможно в десятеричной, в двоичными, в восьмеричной, в шестнадцатеричной системах счисления.

**1.6 Преобразование типов данных**

В языке программирования VKS-2023 возможно преобразование из целочисленного (number) в строковый (str) используя функцию itos(number x). Так же возможно преобразование типа byte в тип number, но нельзя преобразовывать number в byte.

**1.7 Идентификаторы**

Общее количество идентификаторов ограниченно максимальным размером таблицы идентификаторов (4096). Идентификаторы могут содержать символы как нижнего регистра, так и верхнего. Максимальная длина идентификатора равна 20 символам. Идентификаторы функций получают префикс. Префикс состоит из количества параметров функции и сокращение от типов параметров функции (сокращения: i – number и byte, s – str, b – bool, h – htmlobj). Данные правила действуют также для идентификаторов вызова функций. Идентификаторы вызова библиотечных функций являются зарезервированными. Идентификаторы не должны совпадать с ключевыми словами. Типы идентификаторов: имя переменной, имя функции, параметр функции. Имя идентификатора составляется по следующим образом:

* может состоять из цифр, но не может начинаться с цифры;
* состоит из символов латинского алфавита;
* максимальная длина идентификатора равна 20 и не должна превышать это значение. При превышении максимально значения выдаётся ошибка.

**1.8 Литералы**

В языке программирования VKS-2023 существует только 3 типа литералов: целые, символьные и логические. Целочисленные литералы могут быть представлены в виде десятичного, двоичного, шестнадцатеричного, и восьмеричного представления, а строковые литералы – произвольно. Их краткое описание представлено в таблице 1.3..

Таблица 1.3 – Описание литералов

|  |  |
| --- | --- |
| Тип литерала | Описание литерала |
| Целочисленные литералы в десятичном представлении | Последовательность цифр 0…9 с возможным знаком минус |
| Целочисленные литералы в восьмеричном представлении | Последовательность цифр 0…7 с предшествующим двумя символом “8x” |
| Целочисленные литералы в шестнадцатеричном представлении | Последовательность цифр 0…F с предшествующим двумя символом “16x” |
| Целочисленные литералы в двоичном представлении | Последовательность цифр 0 и 1 с предшествующим двумя символом “2x” |

Литерал – числовые значения или строки, представленные в программирование. Целочисленные литералы в десятичном представление – последовательность чисел.

## **1.9 Объявление данных**

Для объявления переменной тип данных и имя идентификатора. Допускается инициализация при объявлении.

Пример объявления числового типа с инициализацией:

**byte** num10 = -109

**number** num2 = 2x1011

**number** num8 = 8x15

**number** num16 = 16x4A

Пример объявления переменной символьного типа с инициализацией:

**str** hello = ‘Привет’

**str** s = ‘car – vehicle’

Пример объявления переменной логического типа с инициализацией:

**bool** tb = true;

Для объявления функций используется ключевое слово **function**, перед которым указывается тип функции. Далее идентификатор, список параметров и тело. Функции типа nodef могут не возвращать значение, остальные должны возвращать значение. Функции могут иметь 0 параметров.

## **1.10 Инициализация данных**

При объявлении переменной допускается инициализация данных. При объявлении переменные инициализируются значением по умолчанию: для строк это пустая строка, для логического false, а для целого это нуль.

Способы инициализации переменных языка программирования VKS-2023 представлены в таблице 1.4.

Таблица 1.4 – Способы инициализации переменных

|  |  |
| --- | --- |
| Вид инициализации | Примечание |
| <тип данных> <идентификатор>; | Автоматическая инициализация переменной number и byte – инициализируется нулем, str– пустой строкой, bool – false. |
| <тип данных> <идентификатор> = <значение>; | Инициализация переменной с присваиванием значения. |

Таким образом, в языке VKS-2023 есть всего 2 вида инициализации переменных

## **1.11 Инструкции языка**

Все возможные инструкции языка программирования VKS-2023 представлены в общем виде в таблице 1.5

Таблица 1.5 – Инструкции языка программирования VKS-2023

|  |  |
| --- | --- |
| Инструкция | Запись |
| Объявление переменной | <тип данных> <идентификатор>;  <тип данных> <идентификатор> = <значение>|<идентификатор>; |
| Присваивание | <идентификатор> = <значение>|<идентификатор>; |
|  |  |

Окончание таблица 1.5

|  |  |
| --- | --- |
| Возврат из подпрограммы | return <выражение>; |
| Условный оператор с блоком else | if (<условие>)  { … }  else  { … } |
| Условный оператор без блока else | if (<условие>)  { … } |
| Цикл с условием | while (<условие>)  { … } |
| Обработчик событий | <идентификатор> even <значение>|<идентификатор>, <идентификатор>([<идентификатор>][, <идентификатор>]\*) |
| Объявление функции | <тип данных> function <идентификатор> ([<тип данных> <идентификатор>][, <тип данных> <идентификатор>]\*) {…} |
| Блок инструкций | { ….  } |

Таким образом язык VKS-2023 использует следующий синтаксис, приведённый выше.

**1.12 Операции языка**

Приоритетность операции умножения выше приоритета операций вычитания и сложения. Для установки наивысшего приоритета используются круглые скобки. К строкам нельзя применять арифметические операции. Язык программирования VKS-2023 может выполнять арифметические и операции сравнения, представленные в таблице 1.6.

Таблица 1.6 – Операции и их приоритеты

|  |  |
| --- | --- |
| Операция | Приоритет операции |
| **=** | 0 |
| == !=  => <=  > < | 1 |
| + - | 2 |
| \* | 3 |

Таким образом, операции в языке VKS-2023 имеют свою приоритетность.

**1.13 Выражения и их вычисления**

Выражения языка VKS-2023 подчиняются следующим правилам:

1. Допускается использовать скобки для смены приоритета операций;
2. Выражение записывается в строку без переносов;
3. Использование двух подряд идущих операторов не допускается;
4. Допускается использовать в выражении вызов функции, вычисляющей и возвращающей значение.

## **1.14 Конструкции языка**

Ключевые программные конструкции языка программирования VKS-2023 представлены в таблице 1.7

Таблица 1.7 – Программные конструкции

|  |  |
| --- | --- |
| Главная функция (точка входа в приложение) | main  {…} |
| Функция | <тип данных> function <идентификатор> ([<тип данных> <идентификатор>][, <тип данных> <идентификатор>]\*)  {  …  } |

Таким образом, данный язык программирования реализует 2 конструкции: главную функцию, которая является точкой входа в программу, и второстепенные функции.

## **1.15 Область видимости идентификаторов**

В языке программирования VKS-2023 переменные обязаны находиться внутри программного блока функций. Область видимости VKS-2023 схожа с лексическим окружением JavaScript. Окружение VKS-2023 содержит в себе идентификаторы переменных, функций и параметров. Каждое окружение ссылается на окружение, в котором было объявлено. В одном окружение не может быть двух одинаковых идентификаторов. Если при обращение к идентификатору в окружение не было найдено объявление идентификатора, то поиск продолжается в родительском окружение. Структура окружения представлена на рисунке 1.2, реализация окружения представлена в приложении Г.

|  |
| --- |
|  |

Рисунок 1.2 – Структура окружения языка программирования VKS-2023

Идентификаторы библиотечных функций заноситься в самое верхнее окружение. Имя для окружения устанавливается идентификатор функции, которой принадлежит это окружение. Если у окружения нету функции, имя объявляется пустой строкой. Окружение высшего уровня имеет имя Global.

**1.16 Семантические проверки**

Перечень семантических проверок, предусмотренных языком, приведен в таблице 1.8.

Таблица 1.8 – Перечень семантических проверок

|  |  |
| --- | --- |
| Номер | Правило |
| 1 | Наличие функции **main** – точки входа в программу; |
| 2 | Единственность точки входа; |
| 3 | Переопределение идентификаторов; |
| 4 | Соответствие сигнатуры функции |
| 5 | Дублирование идентификатора |
| 6 | Использование необъявленного идентификатора |
| 7 | Присваивание значения переменной |
| 8 | Соответствие параметров функции |
| 9 | Совместимость типов |
| 10 | Правильность составленного условного оператора |

Следовательно, при несоблюдении хотя бы одного из правил, семантический анализатор будет сообщать об ошибке.

## **1.17 Распределение оперативной памяти на этапе выполнения**

Так как исходный текст переводится в код на языке JavaScript, то нет необходимости как-либо распределять память.

## **1.18 Стандартная библиотека и ее состав**

В языке есть стандартная библиотека, которая подключается автоматически при трансляции исходного кода в язык ассемблера. Все стандартные библиотеки реализованы на языке JavaScript. Стандартные библиотеки подключены по умолчанию в программу. Вызовы функций доступны в месте вызова пользовательских функций. Содержимое библиотеки и описание функций представлено в таблице 1.9.

Таблица 1.9 – Стандартная библиотека языка VKS-2023

|  |  |
| --- | --- |
| Функция | Характеристика |
| nodef сonsole (strs) | Входной параметр: строка.  Выходной параметр: ничего.  Выводит строку в консоль браузера. |
| number sqrt(number x); | Входной параметр: number.  Выходной параметр: корень из переданного числа. |
| number pow(number x1, number degree); | Входной параметр: 2 числа типа number.  Выходной параметр: возведение, первого передоенного числа (х1), в степень (degree) |
| str concat(str s1, str s2); | Входной параметр: 2 строки типа string.  Выходной параметр: объединение двух строк s1 и s2 |
| number strsize(str s1); | Входной параметр: строка типа str.  Выходной параметр: длина строки. |
| htmlobj CreateVideo(str link); | Входной параметр: строка типа string.  Выходной параметр: возвращает htmlobj, который хранит тэг iframe с ссылкой (link). |
| htmlobj GetHtmlById(str id); | Входной параметр: строка типа str.  Выходной параметр: возвращает htmlobj, который хранит тэг, который имеет заданный id. |
| htmlobj GetBody(); | Входной параметр: строка типа str.  Выходной параметр: возвращает htmlobj, который хранит тэг body. |
| htmlobj Create(str name); | Входной параметр: строка типа str.  Выходной параметр: возвращает htmlobj, который хранит тэг с переданным именем (name). |
| nodef InsertToHtml(htmlobj parent, htmlobj child); | Входной параметр: 2 htmlobj.  Выходной параметр: ничего.  Вставляет в первый преданный тэг (parent), второй переданный тэг (child). |
| nodef TextSet(htmlobj tag, str text); | Входной параметр: htmlobj и строку тпа str.  Выходной параметр: ничего.  Устанавливает в переданном тэге (tag) текст (text) |

Окончание таблицы 1.9

|  |  |
| --- | --- |
| Функция | Характеристика |
| nodef write(htmlobj tag, str text); | Входной параметр: htmlobj и строку тпа str.  Выходной параметр: ничего.  Записывает в переданный тэг (tag) текст (text) |
| nodef writeln(htmlobj tag, str text); | Входной параметр: htmlobj и строку тпа str.  Выходной параметр: ничего.  Записывает с новой строки в переданный тэг (tag) текст (text) |
| nodef SetCSS(htmlobj tag, str css); | Входной параметр: htmlobj и строку тпа str.  Выходной параметр: ничего.  Устанавливает стили для переданного тэга (tag) стили, записанные в параметр css |
| nodef AddCSS(htmlobj tag, str css); | Входной параметр: htmlobj и строку тпа str.  Выходной параметр: ничего.  Добавляет стили для переданного тэга (tag) стили, записанные в параметр css |
| nodef SetAttribute(htmlobj tag, str atr, str value); | Входной параметр: htmlobj и 2 строки тпа str.  Выходной параметр: ничего.  Устанавливает значение атрибут(atr) в value для переданного тэга (tag) |

Большинство библиотечных функцию языка VKS-2023 используются для создания интерактивных html документов.

## **1.20 Точка входа**

Точкой входа является функция main. Точка входа в приложение не может отсутствовать, в программе не может быть больше одной функции main.

**1.21 Препроцессор**

В языке VKS-2023 препроцессор не предусматривается.

## **1.22 Соглашения о вызове**

Исходный код транслируется в язык JavaScript, в котором соглашение о вызове функции не является такой же выразительной концепцией, как в некоторых других языках. Функции вызываются путем использования имени функции и аргументы передаются с помощью объекта arguments, доступный внутри функции.

## **1.23 Объектный код**

Исходный код языка транслируется в язык JavaScript.

**1.24 Классификация сообщений транслятора**

В случае возникновения ошибки в исходном коде программы на языке программирования VKS-2023 и выявлении её транслятором в файл протокола выводится сообщение. Классификация ошибок приведена в таблице 1.10.

Таблица 1.10 – Классификация ошибок

|  |  |
| --- | --- |
| Номера ошибок | Характеристика |
| 0 – 89 | Системные ошибки |
| 90 – 99 | Ошибки лексического анализа |
| 120 – 132 | Ошибки семантического анализа |
| 600 – 608 | Ошибки синтаксического анализа |
| 124-599, 609-999 | Зарезервированные коды ошибок |

В таблице 1.10 представлена классификация ошибок, которые могут возникнуть при анализе исходного кода программы на языке программирования VKS-2023.

**1.25 Контрольный пример**

Контрольный пример представлен в приложении А.

# 2. Структура транслятора

**2.1 Компоненты транслятора, их назначение и принципы взаимодействия**

В языке VKS-2023 исходный код проходит трансляцию в язык JavaScript. Транслятор состоит из нескольких компонентов, которые взаимодействуют между собой и выполняют свои специфические функции, описанные в разделе 2.1. Чтобы получить JavaScript код, требуется использовать выходные данные, сгенерированные лексическим анализатором. Эти данные включают таблицу лексем и таблицу идентификаторов. Для указания выходных файлов, которые будут сгенерированы, используются определенные входные параметры, описанные в таблице. 2.1.

Структура транслятора языка VKS-2023 приведена на рисунке 2.1.

|  |
| --- |
|  |

Рисунок 2.1 – Структура транслятора языка программирования VKS-2023

Первая стадия компиляции называется лексическим анализом, и для ее реализации используется лексический анализатор или сканер. Лексический анализатор принимает последовательность символов исходного языка в качестве входных данных. Он выполняет предварительный анализ текста, преобразуя единый массив символов в отдельные слова или «токены». Примеры таких «токенов» включают идентификаторы, числа, операторы, ключевые слова и другие лексические единицы. Каждой лексеме присваивается тип, и она записывается в таблицу лексем, а также в таблицу идентификаторов, где хранится дополнительная информация. Таблица лексем (ТЛ) и таблица идентификаторов (ТИ) являются входными данными для следующей фазы компиляции - синтаксического анализа или «парсера».

Синтаксический анализатор - это часть компилятора, которая выполняет синтаксический анализ или проверку соответствия исходного кода грамматическим правилам. Входными данными для синтаксического анализатора являются таблица лексем и таблица идентификаторов. Результатом работы синтаксического анализатора является дерево разбора.

Семантический анализатор - это часть транслятора, которая выполняет семантический анализ или проверку исходного кода на наличие ошибок, которые невозможно обнаружить при помощи грамматики. Входными данными для семантического анализатора являются таблица лексем и таблица идентификаторов.

Генератор кода - это часть транслятора, которая создает ассемблерный код на основе данных, полученных на предыдущих этапах компиляции. Входными данными для генератора кода являются таблица лексем и таблица идентификаторов. Генератор кода использует эти данные для создания файла с ассемблерным кодом.

**2.2 Перечень входных параметров транслятора**

Для формирования файлов с результатами работы лексического, синтаксического и семантического анализаторов используются входные параметры транслятора, которые приведены в таблице 2.1.

Таблица 2.1 – Входные параметры транслятора языка VKS-2023

|  |  |  |
| --- | --- | --- |
| Входной параметр | Описание параметра | Значение по умолчанию |
| -in:<путь к in-файлу> | Файл с исходным кодом на языке DML-2021 , имеющий расширение .txt | Не предусмотрено |
| -log:<путь к log-файлу> | Файл журнала для вывода протоколов работы программы | Значение по умолчанию:  <имя in-файла>.log |
| -out:<путь к out-файлу> | Код программы сгенерированный на языке JavaScript | Значение по умолчанию:  <имя in-файла>.js |

В таблице 2.1 представлены входные параметры транслятора языка VKS-2023, которые используются для формирования файлов с результатами работы лексического, синтаксического и семантического анализаторов**.**

**2.3 Протоколы, формируемые транслятором и их содержимое**

В ходе выполнения программы формируются протоколы работы лексического, синтаксического и семантического анализаторов. Таблица с перечнем протоколов, формируемых транслятором языка VKS-2023 и их назначением представлена в таблице 2.2

Таблица 2.2 – Протоколы, формируемые транслятором языка VKS-2023

|  |  |
| --- | --- |
| Формируемый протокол | Описание выходного протокола |
| Файл журнала, заданный параметром "-log:" | Файл с протоколом работы транслятора языка программирования VKS-2023 . Содержит таблицу лексем и таблицу идентификаторов, протокол работы синтаксического анализатора и синтаксического анализа. |
| Выходной файл, c расширением ".js" | Результат работы программы – файл, содержащий исходный код на языке ассемблера. |

Таким образом, при возникновении ошибки, компилятор закончит свою работу, в log-файл будет записана ошибка, а трансляция на JavaScript не будет выполнена.

**3 Разработка лексического анализатора**

**3.1 Структура лексического анализатора**

Первая стадия работы компилятора называется лексическим анализом, а программа, её реализующая, – лексическим анализатором (сканером). На вход лексического анализатора подаётся исходный код входного языка. Лексический анализатор выделяет в этой последовательности простейшие конструкции языка, согласно п. 1.1 стандарта [1]. Лексический анализатор производит предварительный разбор текста, преобразующий единый массив текстовых символов в массив токенов.

Примеры лексических единиц: идентификаторы, числа, символы операций, служебные слова и т.д. Лексический анализатор преобразует исходный текст, заменяя лексические единицы их внутренним представлением – лексемами, для создания промежуточного представления исходной программы. Каждой лексеме сопоставляется ее тип и запись в таблице идентификаторов, в которой хранится дополнительная информация.

Функции лексического анализатора:

* − удаление «пустых» символов и комментариев. Если «пустые» символы (пробелы, знаки табуляции и перехода на новую строку) и комментарии будут удалены лексическим анализатором, синтаксический анализатор никогда не столкнется с ними (альтернативный способ, состоящий в модификации грамматики для включения «пустых» символов и комментариев в синтаксис, достаточно сложен для реализации);
* − распознавание идентификаторов и ключевых слов;
* − распознавание констант;
* − распознавание разделителей и знаков операций.

Исходный код программы представлен в приложении А, структура лексического анализатора представлена на рисунке 3.1.

|  |
| --- |
|  |

Рисунок 3.1 – Структура лексического анализатора

Данная структура позволяет представить визуально работу компилятора.

## **3.2 Входные и выходные данные лексического анализатор**

Входными параметрами являются данные исходного кода. Для удобной работы с исходным кодом, при передаче его в лексический анализатор, все символы разделяются по категориям. В качестве выходных параметров выступают таблица лексем и таблица идентификаторов, которые необходимы для продолжения компиляции, в качестве входных данных синтаксического анализатора.

Структура таблицы контроля входных символов представлена на рисунке 3.2

|  |
| --- |
|  |

Рисунок 3.2. – Таблица контроля входных символов

Так же представлена таблица 3.1 , которая отображает ключевые значения символов для контроля за входными данными

Таблица 3.1 – Соответствие символов и их значений в таблице

|  |  |
| --- | --- |
| Значение в таблице входных символов | Символы |
| Разрешенный | T |
| Запрещенный | F |
| Игнорируемый | I |
| Оператор | О |
| Сепоратор | S |

Выходом лексического анализатора является последовательность токенов.

## **3.3 Параметры лексического анализатора**

Результаты работы лексического анализатора, а именно таблицы лексем и идентификаторов выводятся в файл журнала.

## **3.4 Алгоритм лексического анализа**

* проверяет входной поток символов программы на исходном языке на допустимость, удаляет лишние пробелы и добавляет сепаратор для вычисления номера строки для каждой лексемы;
* для выделенной части входного потока выполняется функция распознавания лексемы;
* при успешном распознавании информация о выделенной лексеме заносится в таблицу лексем и таблицу идентификаторов, и алгоритм возвращается к первому этапу;
* формирует протокол работы;
* при неуспешном распознавании выдается сообщение об ошибке.

Распознавание цепочек основывается на работе конечных автоматов. Работу конечного автомата можно проиллюстрировать с помощью графа переходов. Пример графа для цепочки «else» представлен на рисунке 3.3, где S0 – начальное, а S5 – конечное состояние автомата.



Рисунок 3.3 – Алгоритм работы конечного автомата

Данный пример является достаточно коротким, в реальном языке программирования некоторые графы могут достигать 20-30 состояний.

## **3.5 Контроль входных символов**

Для удобной работы с исходным кодом, при передаче его в лексический анализатор, все символы разделяются по категориям. Таблица входных символов представлена на рисунке 3.2, категории входных символов представлены в таблице 3.1 в разделе 3.2

## **3.6 Удаление избыточных символов**

Лишними символами в данном контексте являются символы табуляции и пробелы, которые не несут смысловой нагрузки в исходном коде программы. Эти излишние символы удаляются в процессе разделения исходного кода на токены.

Алгоритм удаления излишних символов может быть описан следующим образом:

1. Исходный код программы считывается символ за символом.
2. Если встречается пробел или символ табуляции, он рассматривается как разделительный символ.
3. В отличие от других разделительных символов, пробелы и символы табуляции игнорируются.

## **3.7 Перечень ключевых слов**

Соответствие токенов и лексем приведено в таблице 3.2.

Таблица 3.2 – Соответствие токенов и сепараторов с лексемами

|  |  |  |
| --- | --- | --- |
| Токен | Лексема | Пояснение |
| str, byte, number, bool, htmlobj | t | Названия типов данных языка. |
| Идентификатор | i | Длина идентификатора – 20 символов. |
| Литерал | l | Литерал любого доступного типа. |
| function | f | Объявление функции. |
| return | r | Выход из функции. |
| main | m | Главная функция. |
| if | j | Условный оператор if |
| else | e | Условный оператор else |
| ; | ; | Разделение выражений. |
| , | , | Разделение параметров функций. |
| ( | ( | Передача параметров в функцию, приоритет операций. |
| ) | ) | Закрытие блока для передачи параметров, приоритет операций. |
| = | = | Знак присваивания. |
| +, -, \*, / | v | Знаки операций |
| =>, <=, ==, !=, <, > \_ | v | Знаки логических операторов |
| while | w | Условный цикл |

Каждому выражению соответствует детерминированный конечный автомат, по которому происходит разбор данного выражения. Каждый автомат хранится в специальной таблице. Автомат записывается в таблице по индексу кода первого символов. Если есть несколько автоматов с одинаковым начальным символом, то автоматы хранят ссылку на следующие. На автомат, который находиться по индексу соответствующий первому символу лексемы, подаётся токен и с помощью регулярного выражения, соответствующего данному графу переходов, происходит разбор. Если лексема не была распознана, то берётся следующий автомат для распознования. В случае успешного разбора выражения оно записывается в таблицу лексем. Если выражение является идентификатором или литералом, информация также заносится в таблицу идентификаторов. Структура конечного автомата представлен в листинге 3.1.

|  |
| --- |
| struct RELATION  {  char symbol;  short nnode;  RELATION(char c = 0x00, short ns = NULL);  };  struct NODE  {  short n\_relation;  RELATION\* relations;  NODE();  NODE(short n, RELATION rel, ...);  NODE(short n, RELATION \*rel);  };  struct FST {  const char\* str;  short position;  short nstates;  NODE\* nodes;  short\* rstates;  FST();  FST(const char\* s, short ns, NODE n, ...);  FST(const char\* s, short ns, NODE n[]);  }; |

Листинг 3.1 – Структура конечного автомата

Таким образом, конечный автомат рассматривает все возможные реализации токена, при несоответствии ни одному правилу в log-файл подаётся ошибка

## **3.8 Основные структуры данных**

Основными структурами данных лексического анализатора являются таблица лексем и таблица идентификаторов. Таблица лексем содержит номер лексемы, лексему (lexema), полученную при разборе, номер строки в исходном коде (sn), номер в таблице идентификаторов, если лексема является идентификатором (idxTI) и ссылку на операцию, если лексема является операцией. Таблица идентификаторов содержит имя идентификатора (id), номер в таблице лексем (firstApi), тип данных (iddatatype), тип идентификатора (idtype), значение (или параметры функций) (value), ссылку на окружение, в котором был объявлен идентификатор (indEnv) и количество использований идентификатора (refCount).

Код со структурой таблицы лексем представлен в листинге 3.2.

|  |
| --- |
| struct Entry  {  char lexema;  int sn;  int idxTI;  OT::OEntry\* opr;  };  struct LexTable  {  int maxsize;  int size;  Entry\* table;  }; |

Листинг 3.2 Структура таблицы лексем

Код со структурой таблицы идентификаторов представлен на рисунке 3.7.

|  |
| --- |
| struct Entry  {  int s\_number;  int indexPF;  ENV::Environment\* indEnv;  int firstApi;  char id[ID\_MAXSIZE + MAXSUFFIXSIZE + 5];  IDDATATYPE iddatatype;  IDTYPE idtype;  union  {  int vint;  char operation = '\0';  struct  {  unsigned char len;  char s[TI\_S\_MAXSIZE];  } vs;  struct  {  int parmQuantity;  IDDATATYPE type[TI\_MAX\_COUNT\_PARM];  }parm;  } value;  int refCount; |

Листинг 3.3 Структура таблицы идентификаторов

Таким образом, каждый символ проходит по данному алгоритму конечного автомата.

## **3.9 Структура и перечень сообщений лексического анализатора**

В процессе трансляции программы возникают ошибки, которые записываются в протокол. Формат протокола определяется входными параметрами. Если в процессе анализа обнаруживается ошибка, она регистрируется вместе с номером ошибки и диагностическим сообщением, согласно п. 1.1 стандарта [2]. При нахождении ошибки дальнейший анализ прекращается.

## **3.10 Принцип обработки ошибок**

Для обработки ошибок лексический анализатор использует таблицу с сообщениями. Структура сообщений содержит информацию о номере сообщения, номер строки и позицию, где было вызвано сообщение в исходном коде, информацию об ошибке. При возникновении сообщения, лексический анализатор игнорирует найденную ошибку и продолжает работу с исходным кодом, при условии, что найденных ошибок не больше трех. Перечень сообщений представлен в таблице 3.3.

Таблица 3.3 – Сообщения лексического анализатора

|  |  |
| --- | --- |
| Номер ошибки | Описание ошибки |
| 90 | Лексический анализатор. Слово не распознано |
| 91 | Лексический анализатор. Строковый литерал превышает допустимый размер |
| 92 | Лексический анализатор. Строковый литерал не распознан |
| 93 | Лексический анализатор. Идентификатор не распознан |
| 94 | Лексический анализатор. Неверный вид идентификатора |
| 95 | Лексический анализатор. Неизвестный тип переменной |
| 96 | Лексический анализатор. Литерал не распознан |
| 97 | Лексический анализатор. Неизвестная переменная |

Таким образом, при написании кода, достаточно просто понять, с чем конкретно связана та или иная ошибка.

**3.11 Контрольный пример**

Результат работы лексического анализатора в виде таблиц лексем и идентификаторов, соответствующих контрольному примеру, представлен в приложении Б.

**4. Разработка синтаксического анализатора**

**4.1 Структура синтаксического анализатора**

Синтаксический анализатор: часть компилятора, выполняющая синтаксический анализ, то есть исходный код проверяется на соответствие правилам грамматики. Входной информацией для синтаксического анализа является таблица лексем и таблица идентификаторов. Выходной информацией– дерево разбора

Описание структуры синтаксического анализатора языка представлено на рисунке 4.1.

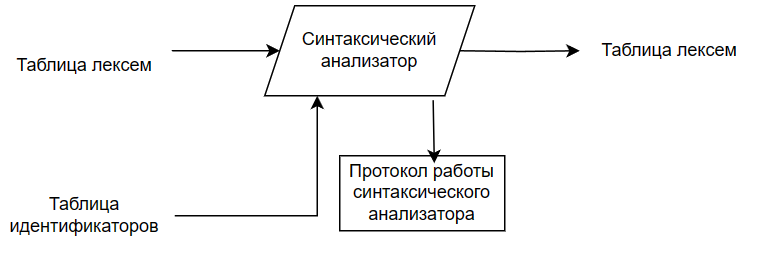


Рисунок 4.1 Структура синтаксического анализатора.

Таким образом, при работе синтаксической части языка программирования VKS-2023, разбираются ключевые ошибки, связанные с последовательностью символов, которые далее передаются в семантический анализатор.

**4.2 Контекстно-свободная грамматика, описывающая синтаксис языка**

В синтаксическом анализаторе транслятора языка VKS-2023 используется контекстно-свободная грамматика , где

T – множество терминальных символов,

N – множество нетерминальных символов,

P – множество правил языка,

S – начальный символ грамматики, являющийся нетерминалом.

Эта грамматика имеет нормальную форму Грейбах, т.к. она не леворекурсивная (не содержит леворекурсивных правил) и правила  имеют вид:

1. , где ; (или , или );
2. , где — начальный символ, при этом если такое правило существует, то нетерминал  не встречается в правой части правил.

Описание нетерминальных символов содержится в таблице 4.1.

Таблица 4.1 – Таблица правил переходов нетерминальных символов

|  |  |  |
| --- | --- | --- |
| Нетерминал | Цепочки правил | Описание |
| S | m{N}  tfiP{N}S | Проверка правильности структуры программы |
| P | (F)  () | Проверка наличия параметров функции |
| F | i  l  i,N  l,N | Проверка на правильность параметров функции при её объявлении |
| U | {}  {}N  {N}  {N}N | Проверка правильности блока |
| I | i  l | Проверка на недопустимое выражение |
| N | iK;  rE;  ti;  ipA;  ipA;N  ti;N  iK;N  i(W);N  tfiPU  j(E)UeU  w(E)U  ti=E;N  ti=E;  i=E;  i=E;N  w(E)U | Проверка операторов |
| K | (W)  () | Проверка на правильность вызова функции |
| E | i  l  (E)  iK  iM  lM  (E)M  iKM | Проверка на правильность арифметического выражения |

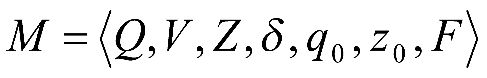
Окончание таблицы 4.1

|  |  |  |
| --- | --- | --- |
| Нетерминал | Цепочки правил | Описание |
| W | i  l  i,W  l,W | Проверка на правильность параметров вызываемой функции |
| M | vE  vEM | Проверка на правильность арифметических действий |
| A | l,iK  i,iK | Проверка на правильность задания обработчика событий |
| K | (W)  () | Проверка на правильность вызова функции |

В таблице 4.1 представлено описание нетерминальных символов и соответствующих правил переходов в контекстно-свободной грамматике языка VKS-2023.

**4.3 Построение конечного магазинного автомата**

Конечный автомат с магазинной памятью представляет собой семерку, согласно п. 1.1 стандарта [3]



Подробное описание компонентов магазинного автомата представлено в таблице 4.2.

Таблица 4.2 – Описание компонентов магазинного автомата

|  |  |  |
| --- | --- | --- |
| Компонента | Определение | Описание |
|  | Множество состояний автомата | Состояние автомата представляет из себя структуру, содержащую позицию на входной ленте, номера текущего правила и цепочки и стек автомата |
|  | Алфавит входных символов | Алфавит представляет из себя множества терминальных и нетерминальных символов, описание которых содержится в таблица 3.1 и 4.1. |
|  | Алфавит специальных магазинных символов | Алфавит магазинных символов содержит стартовый символ и маркер дна стека (представляет из себя символ $) |

Окончание таблицы 4.2

|  |  |  |
| --- | --- | --- |
| Компонента | Определение | Описание |
|  | Начальное состояние автомата | Состояние, которое приобретает автомат в начале своей работы. Представляется в виде стартового правила грамматики |
|  | Начальное состояние магазина автомата | Символ маркера дна стека $ |
|  | Множество конечных состояний | Конечные состояние заставляют автомат прекратить свою работу. Конечным состоянием является пустой магазин автомата и совпадение позиции на входной ленте автомата с размером ленты |
|  | Функция переходов автомата | Функция представляет из себя множество правил грамматики, описанных в таблице 4.1. |

Из данной таблицы можно сделать вывод, что магазинный автомат используется для анализа и обработки языка VKS-2023 с использованием контекстно-свободной грамматики. Автомат состоит из состояний, алфавитов символов, функции переходов и имеет начальное и конечные состояния. С помощью этих компонентов автомат выполняет анализ и трансляцию программного кода на языке VKS-2023.

**4.4 Основные структуры данных**

Основные структуры данных синтаксического анализатора представляются в виде структуры магазинного конечного автомата, выполняющего разбор исходной ленты, и структуры грамматики Грейбах, описывающей синтаксические правила языка VKS-2023. Данные структуры представлены в приложении В.

**4.5 Описание алгоритма синтаксического разбора**

Принцип работы автомата следующий:

* В начале, в магазин записывается стартовый символ.
* На основе полученных таблиц формируется входная последовательность символов, использующаяся в разборе.
* Запускается работа магазинного конечного автомата.
* Автомат берет цепочку, соответствующую текущему нетерминальному символу, и записывает ее инверсию в магазин.
* Если терминалы в стеке и во входной последовательности совпадают – данный терминал удаляется из стека и входной последовательности. Если не совпадают – происходит откат к предыдущему сохраненному состоянию и выбор другой цепочки для текущего нетерминала.
* Если в магазине встречается нетерминальный символ, процесс переходит к пункту 4.
* Если в магазине достигнуто дно, входная последовательность пуста – синтаксический анализ успешно выполнен, иначе генерируется исключение, указывающее на наличие синтаксической ошибки.

**4.6 Параметры синтаксического анализатора и режимы его работы**

Входной информацией для синтаксического анализатора – таблица лексем и идентификаторов. Так же используется описание грамматики в форме Грейбах. Результаты работы лексического разбора: дерево разбора, протокол работы автомата с магазинной памятью выводятся в журнал работы программы.

**4.7 Структура и перечень сообщений синтаксического анализатора**

Перечень сообщений синтаксического анализатора представлены в таблице

4.3.

Таблица 4.3 - Сообщения синтаксического анализатора

|  |  |
| --- | --- |
| Номер ошибки | Описание ошибки |
| 600 | Синтаксическая ошибка. Неверная структура программы |
| 601 | Синтаксическая ошибка. Ошибочный оператор |
| 602 | Синтаксическая ошибка. Ошибка в выражении |
| 603 | Синтаксическая ошибка. Ошибка в операторах выражений |
| 604 | Синтаксическая ошибка. Ошибка в объявление параметров функции |
| 605 | Синтаксическая ошибка. Ошибка в параметрах вызова функции |
| 606 | Синтаксическая ошибка. Ошибка в структуре блока |
| 607 | Синтаксическая ошибка. Ошибка в параметрах функции |
| 608 | Синтаксическая ошибка. Ошибка в вызове параметрах вызова функции |
| 609 | Синтаксическая ошибка. Ошибка в задание параметров обработчика события |

Таким образом, при написании кода, достаточно просто понять, с чем конкретно связана та или иная ошибка.

**4.8 Принцип обработки ошибок**

Синтаксический анализатор выполняет разбор исходной последовательности лексем до тех пор, пока не дойдёт до конца цепочки лексем или не найдёт ошибку. Тогда анализ останавливается и выводится сообщение об ошибке (если она найдена). Если в процессе анализа находятся ошибка, то анализ останавливается.

**4.9 Контрольный пример**

Результаты работы лексического разбора, а именно дерево разбора и протокол работы автомата с магазинной памятью приведены в приложении В.

**5 Разработка семантического анализатора**

**5.1 Структура семантического анализатора**

Семантический анализатор принимает результаты работы лексического и синтаксического анализаторов в качестве входных данных: таблицы лексем, идентификаторов и дерево разбора, полученное в результате синтаксического анализа. Семантический анализатор последовательно проводит различные проверки, чтобы обнаружить возможные ошибки. Однако, некоторые проверки, такие как проверка единственности точки входа или предварительное объявление переменных, могут быть выполнены уже на этапе лексического анализа. Общая структура обособленно семантического анализатора представлена на рисунке 5.1.

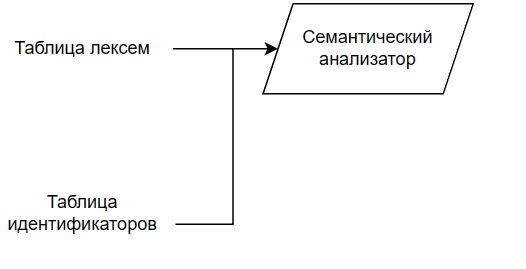


Рисунок 5.1. Структура семантического анализатора

Данная схема отражает то, что семантический анализатор состоит из 2 частей

**5.2 Функции семантического анализатора**

Семантический анализатор выполняет проверку на основные правила языка (семантики языка), которые описаны в разделе 1.16.

**5.3 Структура и перечень сообщений семантического анализатора**

Сообщения семантического анализатора представлены в таблице 5.1.

Таблица 5.1 - Сообщения семантического анализатора

|  |  |
| --- | --- |
| Номер ошибки | Описание ошибки |
| 120 | Семантическая ошибка. Функция объявлена несколько раз |
| 121 | Семантическая ошибка. Переменная объявлена несколько раз |
| 122 | Семантическая ошибка. Превышено максимальное количество аргументов функции |

Окончание таблицы 5.1

|  |  |
| --- | --- |
| Номер ошибки | Описание ошибки |
| 123 | Семантическая ошибка. Типы аргументов не совпадают с типами параметров функции |
| 124 | Семантическая ошибка. Невозможно преобразовать типы данных |
| 125 | Семантическая ошибка. Тип возвращаемого выражения не равен типу функции |
| 126 | Семантическая ошибка. Неизвестная переменная |
| 127 | Семантическая ошибка. Не найден прототип функции |
| 128 | Семантическая ошибка. Переменная не может иметь тип nodef |
| 129 | Семантическая ошибка. Строки не могу использоваться в булевых выражениях |
| 130 | Семантическая ошибка. Ожидалось булевое выражение |
| 131 | Семантическая ошибка. Неверные типы параметров обработчика событий |

Таким образом, при написании кода, достаточно просто понять, с чем конкретно связана та или иная ошибка.

**5.4 Принцип обработки ошибок**

В процессе трансляции программы возникающие ошибки записываются в протокол, который определяется входными параметрами. При возникновении ошибок происходит их протоколирование с указанием номера ошибки и диагностического сообщения. Анализ программы продолжается до нахождения ошибки.

## **5.5 Контрольный пример**

Соответствие примеров некоторых ошибок в исходном коде и диагностических сообщений об ошибках приведено в таблице 5.1.

Таблица 5.1. Примеры диагностики ошибок

|  |  |
| --- | --- |
| Исходный код | Текст сообщения |
| main  {  number x;  strs;  s = x;  } | Ошибка 124 Семантическая ошибка: Типы выражений различны.  Строка 5 |
| number function same()  {  ret 0;  }  number function same()  {  ret 1;  }  main  {  number result = same();  } | Ошибка 120: Семантическая ошибка  Функция объявлена несколько раз.  Строка 6 |

Из приведенных примеров можно сделать вывод, что в процессе трансляции программы на языке VKS-2023 возможны различные семантические ошибки, связанные с отсутствием ключевых слов, несоответствием типов данных и нарушением правил синтаксиса, таких как единственная точка входа. Ошибки записываются в протокол с указанием номера ошибки и диагностического сообщения. Анализ прекращается после нахождения ошибки.

**6. Вычисление выражений**

**6.1 Выражения, допускаемые языком**

В языке VKS-2023 допускаются вычисления выражений целочисленного типа данных с поддержкой вызова функций внутри выражений. Приоритет операций представлен на таблице 6.1.

Таблица 6.1. Приоритеты операций

|  |  |
| --- | --- |
| Операция | Значение приоритета |
| 1 | > < => <= != == |
| 2 | + - |
| 3 | \* |

Из таблицы 6.1 в языке VKS-2023 можно сделать вывод, что при вычислении выражений целочисленного типа данных следует учитывать порядок операций, определенный в таблице. Операторы в скобках имеют наивысший приоритет, за ними следуют умножение, за ними сложение и вычитание, а затем логические операторы.

**6.2 Польская запись и принцип её построения**

В языке VKS-2023 выражения не преобразуются к польской записи, так как исходный код транслируется в JavaScript.

**6.3 Программная реализация обработки выражений**

В языке VKS-2023 не реализована польская запись.

**6.4 Контрольный пример**

В языке VKS-2023 не реализована польская запись.

**7. Генерация кода**

**7.1 Структура генератора кода**

В языке VKS-2023 генерация кода является последним этапом процесса трансляции. Генератор кода принимает таблицы лексем и идентификаторов, которые были получены в результате лексического анализа. На основе этих таблиц генератор создает выходной файл на языке JavaScript, который представляет собой окончательный результат работы транслятора. В случае возникновения ошибок в процессе трансляции, генерация кода не будет выполнена.

**7.2 Представление типов данных в оперативной памяти**

Тип данных bool в JavaScript является типом boolen и занимает 1 байт. Тип данных number, byte в JavaScript является типом number и занимает 8 байт. Тип str в JavaScript каждый символ занимает 16 бит. Данные 4 типа хранятся в стеке памяти. Тип данных htmlobj в языке JavaScript является объектом и может иметь произвольный размер. Данный тип храниться в кучу.

**7.3 Статическая библиотека**

В языке VKS-2023 предусмотрена статическая библиотека. Статическая библиотека содержит функции, написанные на языке JavaScript. Идентификаторы функций находятся в глобальном окружении. Функции автоматически подключатся при использовании.

**7.4 Особенности алгоритма генерации кода**

В языке VKS-2023 генерация кода строится на основе таблиц лексем и идентификаторов.

|  |
| --- |
|  |

Рисунок 7.1 Структура генератора кода

Таким данный генератор кода получает дынные из таблицы лексем и таблицы идентификаторов, применяет к ним функции преобразующие код на JavaScript и записывает его в файл .js.

**7.5 Параметры, управляющие генерацией кода**

На вход генератору кода поступают таблицы лексем и идентификаторов исходного кода программы на языке VKS-2023. Результаты работы генератора кода выводятся в файл с расширением .js.

## **7.6 Контрольный пример**

Результат генерации JavaScript кода на основе контрольного примера из приложения А приведен в приложении Д. Результат работы контрольного примера приведён на рисунке 7.2.

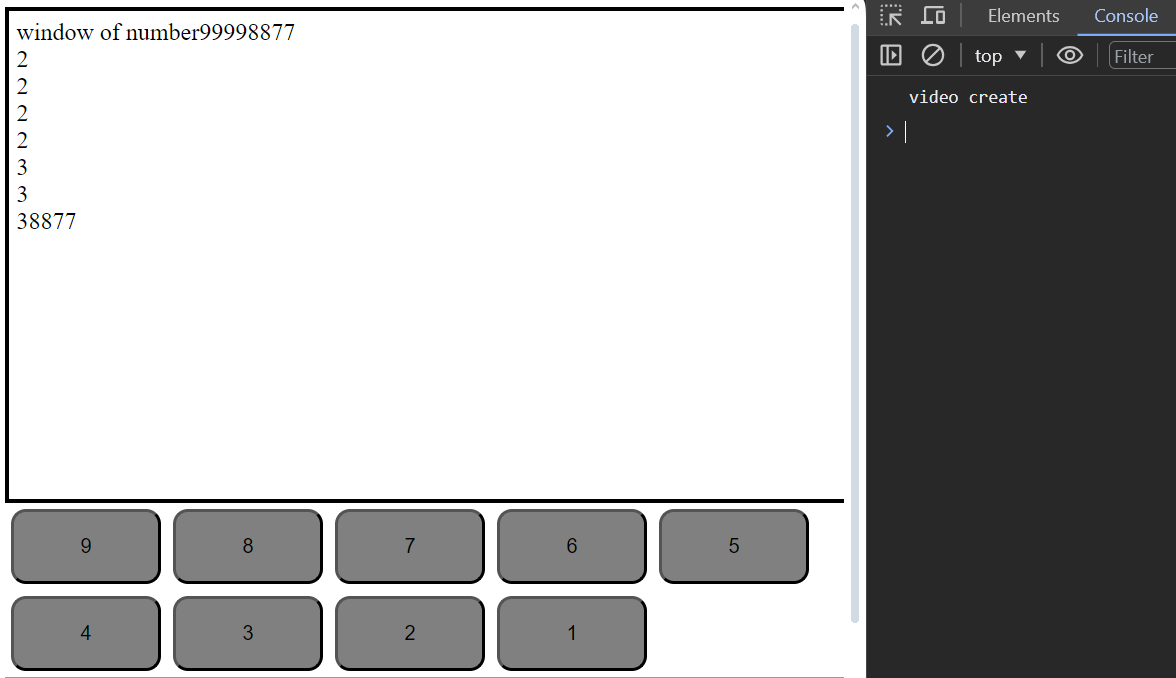


Рисунок 7.2. Результат работы скрипта на языке VKS-2023

Данный пример хорошо отображает всю суть программирования на языке VKS-2023, показывая реализацию ключевого функционала.

**8. Тестирование транслятора**

**8.1 Общие положения**

Тестирование должно покрывать как можно больше сценариев использования языка и его конструкций. Все тесты были представлены для типичных ошибок пользователей при использовании языка. Когда компилятор обнаруживает ошибку, он записывает информацию о ней в протокол, содержащий номер ошибки и диагностическое сообщение, помогающее разработчику понять причину ошибки компиляции. Результаты тестирования записываются в файл .log.

**8.2 Результаты тестирования**

В языке программирования VKS-2023 не разрешается использовать запрещенные входным алфавитом символы. Результат использования запрещенного символа показан в таблице 8.1.

Таблица 8.1 – Тестирование фазы проверки на допустимость символов

|  |  |
| --- | --- |
| Исходный код | Диагностическое сообщение |
| str function Ђfelev{ret 'sr';} | Ошибка 111: Недопустимый символ в исходном файле (-in)  Строка 1 позиция 9 |

На этапе лексического анализа в языке VKS-2023 могут возникнуть ошибки, описанные в пункте 3.7. Результаты тестирования лексического анализатора показаны в таблице 8.2.

Таблица 8.2 - Тестирование лексического анализатора

|  |  |
| --- | --- |
| Исходный код | Диагностическое сообщение |
| main  {  number вап;  } | Ошибка 98: Лексический анализатор: Лексема не распознана. Строка 3 позиция 12 |

На этапе синтаксического анализа в языке VKS-2023 могут возникнуть ошибки, описанные в пункте 4.6. Результаты тестирования синтаксического анализатора показаны в таблице 8.3.

Таблица 8.3 - Тестирование синтаксического анализатора

|  |  |
| --- | --- |
| Исходный код | Диагностическое сообщение |
| fun(x,y, +, z); | Ошибка 605: Синтаксическая ошибка. Ошибка в параметрах вызова функции, строка 23 |

Семантический анализ в языке VKS-2023 содержит множество проверок по семантическим правилам, описанным в пункте 1.16. Итоги тестирования семантического анализатора на корректное обнаружение семантических ошибок приведены в таблице 8.4.

Таблица 8.4 - Тестирование семантического анализатора

|  |  |
| --- | --- |
| Исходный код | Диагностическое сообщение |
| main  {  number x;  str s;  s = x;  } | Ошибка 124: Семантическая ошибка. Типы выражения различны.  Строка 5 |
| strfunction fi(strs)  {  return s;  }  main  {  number x;  str s;  s = fi(x);  } | Ошибка 127: Семантическая ошибка. Не найден прототип функции.  Строка 9 |
| strfunction fi(str s)  {  return s;  }  number function fi(str s)  {  return s;  }  main  {  number x;  str s;  s = fi(s);  } | Ошибка 120: Семантическая ошибка. Функция объявлена несколько раз.  Строка 9 |

Таким образом данный раздел предоставляет набор тестов для проверки лексического, синтаксического и семантического анализаторов.

# Заключение

В ходе выполнения курсовой работы был разработан транслятор и генератор кода для языка программирования VKS-2023. В результате были выполнены следующие задачи:

1. Была сформулирована спецификация языка VKS-2023, в которой определены его основные элементы и правила.
2. Были разработаны конечные автоматы и алгоритмы для эффективной работы лексического анализатора, который распознает допустимые цепочки символов языка.
3. Была реализована программная версия лексического анализатора.
4. Была разработана контекстно-свободная грамматика для описания синтаксически верных конструкций языка.
5. Была реализована программная версия синтаксического анализатора.
6. Был разработан семантический анализатор, который проверяет используемый код на соответствие логическим правилам.
7. Был разработан транслятор кода на язык ассемблера.
8. Было проведено тестирование всех компонентов транслятора.

Окончательная версия языка VKS-2023 включает следующие возможности:

1. 2 типа данных: беззнаковое целое и строковый.
2. Поддержка операторов ввода и вывода.
3. 4 арифметических оператора для вычисления выражений.
4. Шесть логических операторов для использования в условной конструкции.
5. Поддержка функций и условий.
6. Библиотека стандартных функций языка.
7. Система обработки ошибок пользователя.

В результате работы были получены следующие результаты:

1. Было получено представление о структурах и процессах, используемых при построении трансляторов.
2. Были изучены основные различия и преимущества тех или иных средств трансляции.

# Список использованных источников

1 Studfiles – Лексический анализатор: проектирование, принципы построения и реализации [Электронный ресурс]. – Режим доступа: https://studfile.net/preview/6854883/ – Дата доступа: 01.12.2023.

2. Habr – Об изучении компиляторов и создании языков программирования [Электронный ресурс]. – Режим доступа: https://habr.com/ru/search/. – Дата доступа: 28.11.2023.

3. Studfiles – Алгоритмы синтаксического разбора [Электронный ресурс]. – Режим доступа: https://studfile.net/preview/6854884/ – Дата доступа: 29.11.2023.

4. Герберт, Ш. Справочник программиста по C/C++ / Шилдт Герберт. - 3-е изд. – Москва : Вильямс, 2003. - 429 с.

5. Прата, С. Язык программирования С++. Лекции и упражнения / С. Прата. – М., 2006 — 1104 c.

# Приложение А

|  |
| --- |
| htmlobj function Create(str name, str css)  {  byte usl = -22;  htmlobj tag = Create(name);  SetCSS(tag, css);  return tag;  }  htmlobj function CreateButton(str value)  {  htmlobj button = Create('input', 'width: 100px; height: 50px; margin: 4px; background-color: gray; border-radius: 10px; ');  SetAttribute(button, 'type', 'button');  SetAttribute(button, 'value', value);  return button;  }  main  {  htmlobj body = GetBody();  htmlobj window = Create('div', 'width: 560px; height: 315px; background-color: white; box-sizing: content-box; padding: 5px; border: solid 3px black;');  InsertToHtml(body, window);  nodef function YoutubeVidoMaker(str shortLink)  {  str link = concat(shortLink, '?autoplay=1&loop=1');  htmlobj videoTag = CreateVideo(link);  InsertToHtml(body, videoTag);  }  TextSet(window, 'window of number');  byte i = 9;  while(i != 0)  {  str s = itos(i);  htmlobj button = CreateButton(s);  InsertToHtml(body, button);  if(i <= 4)  {  button event 'click', writeln(window, '');  }  button event 'click', write(window, s);  i = i -1;  }    YoutubeVidoMaker('https://www.youtube.com/embed/X-ANZ8ba8jU');  } |

Листинг 1 – Исходный код на языке VKS-2023

# Приложение Б

------------------ Литералы ------------------

Идентификатор: Тип данных: Значение: Длина строки: Первое вхождение:

22 l\_0 BYTE -22 - 15

27 l\_1 STR 'input' 7 48

28 l\_2 STR 'width: 100px; height: 50px; margin: 4px; background-color: gray; border-radius: 10px; ' 88 50

29 l\_3 STR 'type' 6 57

30 l\_4 STR 'button' 8 59

31 l\_5 STR 'value' 7 66

35 l\_6 STR 'div' 5 89

36 l\_7 STR 'width: 560px; height: 315px; background-color: white; box-sizing: content-box; padding: 5px; border: solid 3px black;' 119 91

40 l\_8 STR '?autoplay=1&loop=1' 20 116

42 l\_9 STR 'window of number' 18 139

44 l\_10 BYTE 9 - 145

45 l\_11 BYTE 0 - 151

48 l\_12 BYTE 4 - 181

49 l\_13 STR 'click' 7 186

50 l\_14 STR '' 2 192

51 l\_15 STR 'click' 7 198

52 l\_16 BYTE 1 - 211

53 l\_17 STR 'https://www.youtube.com/embed/X-ANZ8ba8jU' 43 216

------------------ Функции ------------------

Идентификатор: Тип данных возврата: Количество переданных параметров: Первое вхождение: параметры

0 console1s NODEF 1 -1 STR

1 sqrt1i INT 1 -1 INT

2 pow2ii INT 2 -1 INT INT

3 concat2ss STR 2 -1 STR STR

4 itos1i STR 1 -1 INT

5 strsize1s INT 1 -1 STR

6 CreateVideo1s HTMLOBJ 1 -1 STR

7 tomas0 NODEF 0 -1

8 GetHtmlById1s HTMLOBJ 1 -1 STR

9 GetBody0 HTMLOBJ 0 -1

10 Create1s HTMLOBJ 1 -1 STR

11 InsertToHtml2hh NODEF 2 -1 HTMLOBJ HTMLOBJ

12 TextSet2hs NODEF 2 -1 HTMLOBJ STR

13 write2hs NODEF 2 -1 HTMLOBJ STR

14 writeln2hs NODEF 2 -1 HTMLOBJ STR

15 SetCSS2hs NODEF 2 -1 HTMLOBJ STR

16 AddCSS2hs NODEF 2 -1 HTMLOBJ STR

17 SetAttribute3hss NODEF 3 -1 HTMLOBJ STR STR

18 Create2ss HTMLOBJ 2 2 STR STR

24 CreateButton1s HTMLOBJ 1 37 STR

32 main0 NODEF 0 75

37 YoutubeVidoMaker1s NODEF 1 103 STR

------------------ Переменные ------------------

Имя окружения: Идентификатор: Тип данных: Тип идентификатора: Первое вхождение:

19 Create name STR P 5

20 Create css STR P 8

21 Create usl BYTE V 12

23 Create tag HTMLOBJ V 17

25 CreateButton value STR P 40

26 CreateButton button HTMLOBJ V 44

33 main body HTMLOBJ V 78

34 main window HTMLOBJ V 85

38 YoutubeVidoMaker shortLink STR P 106

39 YoutubeVidoMaker link STR V 110

41 YoutubeVidoMaker videoTag HTMLOBJ V 120

43 main i BYTE V 143

46 s STR V 155

47 button HTMLOBJ V 163

Листинг 1 – Таблица идентификаторов контрольного примера

|  |
| --- |
| 1 tfi(ti,ti)  2 {  3 ti=l;  4 ti=i(i);  5 i(i,i);  6 ri;  7 }  8  9 tfi(ti)  10 {  11 ti=i(l,l);  12 i(i,l,l);  13 i(i,l,i);  14 ri;  15 }  16  17  18 m  19 {  20 ti=i();  21 ti=i(l,l);  22 i(i,i);  23 tfi(ti)  24 {  25 ti=i(i,l);  26 ti=i(i);  27 i(i,i);  28 }  29 i(i,l);  30 ti=l;  31 w(ivl)  32 {  33 ti=i(i);  34 ti=i(i);  35 i(i,i);  36 j(ivl)  37 {  38 ipl,i(i,l);  39 }  40 ipl,i(i,i);  41 i=ivl;  42 }  43  44 i(l);  45 } |

Листинг 2 – Таблица лексем контрольного примера

# Приложение В

|  |
| --- |
| #pragma once  #include "GRB.h"  #define GRB\_ERROR\_SERIES 600  namespace GRB  {  #define NS(n) GRB::Rule::Chain::N(n)  #define TS(n) GRB::Rule::Chain::T(n)  Greibach greibach(NS('S'), TS('$'),  10,  Rule(NS('S'), GRB\_ERROR\_SERIES + 0, // Синтаксическая ошибка. Неверная структура программы  2,  Rule::Chain(4, TS('m'), TS('{'), NS('N'), TS('}')),  //Rule::Chain(9, TS('t'), TS('f'), TS('i'), TS('('), NS('F'), TS(')'), TS('{'), NS('N'), TS('}')),  Rule::Chain(8, TS('t'), TS('f'), TS('i'), NS('P'), TS('{'), NS('N'), TS('}'), NS('S'))  //Rule::Chain(13, TS('t'), TS('f'), TS('i'), TS('('), NS('F'), TS(')'), TS('{'), NS('N'), TS('r'), NS('E'), TS(';'), TS('}'), NS('S'))  //Rule::Chain(15, TS('t'), TS('f'), TS('i'), TS('('), NS('F'), TS(')'), TS('{'), NS('N'), TS('d'), NS('S'), TS('r'), NS('E'), TS(';'), TS('}'), NS('S'))  ),  Rule(NS('N'), GRB\_ERROR\_SERIES + 1, // Синтаксическая ошибка. Ошибочный оператор  15,  Rule::Chain(3, TS('i'), NS('K'), TS(';')),  Rule::Chain(3, TS('t'), TS('i'), TS(';')),  Rule::Chain(3, TS('r'), NS('E'), TS(';')),  Rule::Chain(4, TS('i'), TS('p'), NS('A'), TS(';')),  Rule::Chain(4, TS('i'), TS('='), NS('E'), TS(';')),  Rule::Chain(5, TS('t'), TS('i'), TS('='), NS('E'), TS(';')),  Rule::Chain(4, TS('i'), NS('K'), TS(';'), NS('N')),  Rule::Chain(4, TS('t'), TS('i'), TS(';'), NS('N')),  Rule::Chain(5, TS('i'), TS('p'), NS('A'), TS(';'), NS('N')),  Rule::Chain(5, TS('w'), TS('('), NS('E'), TS(')'), NS('U')),  Rule::Chain(5, TS('j'), TS('('), NS('E'), TS(')'), NS('U')),  Rule::Chain(5, TS('i'), TS('='), NS('E'), TS(';'), NS('N')),  Rule::Chain(6, TS('t'), TS('i'), TS('='), NS('E'), TS(';'), NS('N')),  Rule::Chain(7, TS('j'), TS('('), NS('E'), TS(')'), NS('U'), TS('e'), NS('U')),  Rule::Chain(5, TS('t'), TS('f'), TS('i'), NS('P'), NS('U'))    ),  Rule(NS('E'), GRB\_ERROR\_SERIES + 2, //Синтаксическая ошибка. Ошибка в выражении  8,  Rule::Chain(1, TS('i')),  Rule::Chain(1, TS('l')),  Rule::Chain(3, TS('('), NS('E'), TS(')')),  Rule::Chain(2, TS('i'), NS('K')),  Rule::Chain(2, TS('i'), NS('M')),  Rule::Chain(2, TS('l'), NS('M')),  Rule::Chain(3, TS('i'), NS('K'), NS('M')),  Rule::Chain(4, TS('('), NS('E'), TS(')'), NS('M'))  ),  Rule(NS('M'), GRB\_ERROR\_SERIES + 3, // Синтаксическая ошибка. Ошибка в операторах выражений  2,  Rule::Chain(2, TS('v'), NS('E')),  Rule::Chain(3, TS('v'), NS('E'), NS('M'))  ),  Rule(NS('F'), GRB\_ERROR\_SERIES + 4, // Синтаксическая ошибка. Ошибка в параметрах функции  2,  //Rule::Chain(0, TS('t')),  Rule::Chain(2, TS('t'), TS('i')),  Rule::Chain(4, TS('t'), TS('i'), TS(','), NS('F'))  ),  Rule(NS('W'), GRB\_ERROR\_SERIES + 5, // Синтаксическая ошибка. Ошибка в параметрах вызываемой функции  4,  Rule::Chain(1, TS('i')),  Rule::Chain(1, TS('l')),  Rule::Chain(3, TS('i'), TS(','), NS('W')),  Rule::Chain(3, TS('l'), TS(','), NS('W'))  ),  Rule(NS('U'), GRB\_ERROR\_SERIES + 6, // Синтаксическая ошибка. Ошибка в струторе блока  4,  Rule::Chain(2, TS('{'), TS('}')),  Rule::Chain(3, TS('{'), TS('}'), NS('N')),  Rule::Chain(3, TS('{'), NS('N'), TS('}')),  Rule::Chain(4, TS('{'), NS('N'), TS('}'), NS('N'))  ),  Rule(NS('P'), GRB\_ERROR\_SERIES + 7, // Синтаксическая ошибка. Ошибка в парметрах функции  2,  Rule::Chain(2, TS('('), TS(')')),  Rule::Chain(3, TS('('), NS('F'), TS(')'))  ),  Rule(NS('K'), GRB\_ERROR\_SERIES + 8, // Синтаксическая ошибка. Ошибка в вызове функции  2,  Rule::Chain(2, TS('('), TS(')')),  Rule::Chain(3, TS('('), NS('W'), TS(')'))  ),  Rule(NS('A'), GRB\_ERROR\_SERIES + 9, // Синтаксическая ошибка. Ошибка в задание параметров оброботчика события  2,  Rule::Chain(4, TS('l'), TS(','), TS('i'), NS('K')),  Rule::Chain(4, TS('i'), TS(','), TS('i'), NS('K'))  )  );  } |

Листинг 1 – Грамматика языка VKS-2023

|  |
| --- |
| struct MfstState // состояние автомата (для сохранения)  {  short lenta\_position; // позиция на ленте  short nrule; // номер текущего правила  short nrulechain; // номер текущей цепочки, текущего правила  MFSTSTACK st; // стек автомата  MfstState();  MfstState(  short pposition, // позиция на ленте  MFSTSTACK pst, // стек автомата  short pnrulechain // номер текущей цепочки, текущего правила  );  MfstState(  short pposition, // позиция на ленте  MFSTSTACK pst, // стек автомата  short pnrule, // номер текущего правила  short pnrulechain // номер текущей цепочки, текущего правила  );  };  struct Mfst // магазинный автомат  {  enum RC\_STEP { // код возвтара функции step  NS\_OK, // найдено правило и цепочка, цепочка записана в стек  NS\_NORULE, // не найдено правило грамматики (ошибка в грамматике)  NS\_NORULECHAIN, // не найдена подходящая цепочка правила (ошибка в исходном коде)  NS\_ERROR, // неизвестный нетерминальный символ грамматики  TS\_OK, // тек. символ ленты == вершине стека, продвинулась лента, pop стека  TS\_NOK, // тек. символ ленты != вершине стека, восстановлено состояние  LENTA\_END, // текущая позиция ленты >= lenta\_size  SURPRISE // неожиданный код возврата (ошибка в step)  };  struct MfstDiagnosis // диагностика  {  short lenta\_position; // позиция на ленте  RC\_STEP rc\_step; // код завершения шага  short nrule; // номер правила  short nrule\_chain; // номер цепочки правила  MfstDiagnosis();  MfstDiagnosis(  short plenta\_position, // позиция на ленте  RC\_STEP prt\_step, // код завершения шага  short pnrule, // номер правила  short pnrule\_chain // номер цепочки правила  );  } diagnosis[MFST\_DIAGN\_NUMBER]; // последние самые глубокие сообщения  GRBALPHABET\* lenta; // перекодированная (TS/NS) лента (из LEX)  short lenta\_position; // текущая позиция на ленте  short nrule; // номер текущего правила  short nrulechain; // номер текущей цепочки, текущего правила  short lenta\_size; // размер ленты  GRB::Greibach grebach; // грамматика Грейбах  LEXA::Tables lex; // результат работы лексического анализатора  MFSTSTACK st; // стек автомата  std::stack<MfstState> storestate; // стек для сохранения состояний  Mfst();  Mfst(  LEXA::Tables plex, // результат работы лексического анализатора  GRB::Greibach pgrebach // грамматика Грейбах  );  char\* getCSt(char\* buf); // получить содержимое стека  char\* getCLenta(char\* buf, short pos, short n = 25); // лента: n символов с pos  char\* getDiagnosis(short n, char\* buf); // получить n-ю строку диагностики или 0х00  bool savestate(); // сохранить состояние автомата  bool reststate(); // восстановить состояние автомата  bool push\_chain( // поместить цепочку правила в стек  GRB::Rule::Chain chain // цепочка правила  );  RC\_STEP step(); // выполнить шаг автомата  bool start(Log::LOG &logref); // запустить автомат  bool savediagnosis(  RC\_STEP pprc\_step // код завершения шага  );  void printrules(); // вывести последовательность правил  struct Deducation // вывод  {  short size; // количество шагов в выводе  short\* nrules; // номера правил грамматики  short\* nrulechains; // номера цепочек правик грамматики (nrules)  Deducation() { size = 0; nrules = 0; nrulechains = 0; };  } deducation;  bool savededucation(); // сохранить дерево вывода  }; |

Листинг 2 – Структура магазинного автомата

|  |
| --- |
| #pragma once  #include "Error.h"  typedef short GRBALPHABET; // символы алфавита грамматики терминалы > 0, нетерминалы < 0  namespace GRB  {  struct Rule // правило в грамматике Грейбах  {  GRBALPHABET nn; // нетерминал (левый символ правила) < 0  int iderror; // идентификатор диагностического сообщени¤  short size; // количество цепочек - правых частей правила  struct Chain // цепочка (правая часть правила)  {  short size; // длина цепочки  GRBALPHABET\* nt; // цепочка терминалов (> 0) и нетерминалов (< 0)  Chain() { size = 0; nt = 0; };  Chain(  short psize, // количество символов в цепочке  GRBALPHABET s, ... // символы (терминал или нетерминал)  );  char\* getCChain(char\* b); // получить правую сторону правила  static GRBALPHABET T(char t) { return GRBALPHABET(t); }; // терминал  static GRBALPHABET N(char n) { return -GRBALPHABET(n); }; // нетерминал  static bool isT(GRBALPHABET s) { return s > 0; }; // терминал?  static bool isN(GRBALPHABET s) { return !isT(s); }; // нетерминал?  static char alphabet\_to\_char(GRBALPHABET s) { return isT(s) ? char(s) : char(-s); }; // GRBALPHABET->char  }\*chains; // массив цепочек - правых частей правила  Rule() { nn = 0x00; size = 0; };  Rule(  GRBALPHABET pnn, // нетерминал (< 0)  int iderror, // идентификатор диагностического сообщения (Error)  short psize, // количество цепочек - правых частей правила  Chain c, ... // множество цепочек - правых частей правила  );  char\* getCRule( // получить правило в виде N-цепочки (для распечатки)  char\* b, // буфер  short nchain // номер цепочки (правой части) в правиле  );  short getNextChain( // получить следующую за j подход¤щую цепочку, вернуть её номер или -1  GRBALPHABET t, // первый символ цепочки  Rule::Chain& pchain, // возвращаемая цепочка  short j // номер цепочки  );  };  struct Greibach // грамматика Грейбах  {  short size; // количество правил  GRBALPHABET startN; // стартовый символ  GRBALPHABET stbottomT; // дно стека  Rule\* rules; // множество правил  Greibach() { short size = 0; startN = 0; stbottomT = 0; rules = 0; };  Greibach(  GRBALPHABET pstartN, // стартовый символ  GRBALPHABET pstbottomT, // дно стека  short psize, // количество правил  Rule r, ... // правила  );  short getRule( // получить правило, возвращающа¤ номер правила или -1  GRBALPHABET pnn, // левый символ  Rule& prule // возвращаемое правило грамматики  );  Rule getRule(short n); // получить правило по номеру  };  Greibach getGreibach(); // получить грамматику  }  struct Greibach // грамматика Грейбах  {  short size; // количество правил  GRBALPHABET startN; // стартовый символ  GRBALPHABET stbottomT; // дно стека  Rule\* rules; // множество правил  Greibach() { short size = 0; startN = 0; stbottomT = 0; rules = 0; };  Greibach(  GRBALPHABET pstartN, // стартовый символ  GRBALPHABET pstbottomT, // дно стека  short psize, // количество правил  Rule r, ... // правила  );  short getRule( // получить правило, возвращающа¤ номер правила или -1  GRBALPHABET pnn, // левый символ  Rule& prule // возвращаемое правило грамматики  );  Rule getRule(short n); // получить правило по номеру  }; |

Листинг – 3 Структура грамматики Грейбах

Шаг : Правило Входная лента Стек

0 : S->tfiP{N}S tfi(ti,ti){ti=l;ti=i(i);i S$

0 : SAVESTATE: 1

0 : tfi(ti,ti){ti=l;ti=i(i);i tfiP{N}S$

1 : fi(ti,ti){ti=l;ti=i(i);i( fiP{N}S$

2 : i(ti,ti){ti=l;ti=i(i);i(i iP{N}S$

3 : (ti,ti){ti=l;ti=i(i);i(i, P{N}S$

4 : P->() (ti,ti){ti=l;ti=i(i);i(i, P{N}S$

4 : SAVESTATE: 2

4 : (ti,ti){ti=l;ti=i(i);i(i, (){N}S$

5 : ti,ti){ti=l;ti=i(i);i(i,i ){N}S$

6 : TS\_NOK/NS\_NORULECHAIN

6 : RESSTATE

6 : (ti,ti){ti=l;ti=i(i);i(i, P{N}S$

7 : P->(F) (ti,ti){ti=l;ti=i(i);i(i, P{N}S$

7 : SAVESTATE: 2

7 : (ti,ti){ti=l;ti=i(i);i(i, (F){N}S$

8 : ti,ti){ti=l;ti=i(i);i(i,i F){N}S$

9 : F->ti ti,ti){ti=l;ti=i(i);i(i,i F){N}S$

9 : SAVESTATE: 3

9 : ti,ti){ti=l;ti=i(i);i(i,i ti){N}S$

10 : i,ti){ti=l;ti=i(i);i(i,i) i){N}S$

11 : ,ti){ti=l;ti=i(i);i(i,i); ){N}S$

12 : TS\_NOK/NS\_NORULECHAIN

12 : RESSTATE

12 : ti,ti){ti=l;ti=i(i);i(i,i F){N}S$

13 : F->ti,F ti,ti){ti=l;ti=i(i);i(i,i F){N}S$

13 : SAVESTATE: 3

13 : ti,ti){ti=l;ti=i(i);i(i,i ti,F){N}S$

14 : i,ti){ti=l;ti=i(i);i(i,i) i,F){N}S$

15 : ,ti){ti=l;ti=i(i);i(i,i); ,F){N}S$

16 : ti){ti=l;ti=i(i);i(i,i);r F){N}S$

17 : F->ti ti){ti=l;ti=i(i);i(i,i);r F){N}S$

17 : SAVESTATE: 4

17 : ti){ti=l;ti=i(i);i(i,i);r ti){N}S$

18 : i){ti=l;ti=i(i);i(i,i);ri i){N}S$

19 : ){ti=l;ti=i(i);i(i,i);ri; ){N}S$

20 : {ti=l;ti=i(i);i(i,i);ri;} {N}S$

21 : ti=l;ti=i(i);i(i,i);ri;}t N}S$

22 : N->ti; ti=l;ti=i(i);i(i,i);ri;}t N}S$

22 : SAVESTATE: 5

22 : ti=l;ti=i(i);i(i,i);ri;}t ti;}S$

23 : i=l;ti=i(i);i(i,i);ri;}tf i;}S$

24 : =l;ti=i(i);i(i,i);ri;}tfi ;}S$

25 : TS\_NOK/NS\_NORULECHAIN

25 : RESSTATE

25 : ti=l;ti=i(i);i(i,i);ri;}t N}S$

26 : N->ti=E; ti=l;ti=i(i);i(i,i);ri;}t N}S$

26 : SAVESTATE: 5

26 : ti=l;ti=i(i);i(i,i);ri;}t ti=E;}S$

27 : i=l;ti=i(i);i(i,i);ri;}tf i=E;}S$

2535: ivl;}i(l);} E;}N}$

2536: E->iK ivl;}i(l);} E;}N}$

2536: SAVESTATE: 106

2536: ivl;}i(l);} iK;}N}$

2537: vl;}i(l);} K;}N}$

2538: TNS\_NORULECHAIN/NS\_NORULE

2538: RESSTATE

2538: ivl;}i(l);} E;}N}$

2539: E->iM ivl;}i(l);} E;}N}$

2539: SAVESTATE: 106

2539: ivl;}i(l);} iM;}N}$

2540: vl;}i(l);} M;}N}$

2541: M->vE vl;}i(l);} M;}N}$

2541: SAVESTATE: 107

2541: vl;}i(l);} vE;}N}$

2542: l;}i(l);} E;}N}$

2543: E->l l;}i(l);} E;}N}$

2543: SAVESTATE: 108

2543: l;}i(l);} l;}N}$

2544: ;}i(l);} ;}N}$

2545: }i(l);} }N}$

2546: i(l);} N}$

2547: N->iK; i(l);} N}$

2547: SAVESTATE: 109

2547: i(l);} iK;}$

2548: (l);} K;}$

2549: K->() (l);} K;}$

2549: SAVESTATE: 110

2549: (l);} ();}$

2550: l);} );}$

2551: TS\_NOK/NS\_NORULECHAIN

2551: RESSTATE

2551: (l);} K;}$

2552: K->(W) (l);} K;}$

2552: SAVESTATE: 110

2552: (l);} (W);}$

2553: l);} W);}$

2554: W->l l);} W);}$

2554: SAVESTATE: 111

2554: l);} l);}$

2555: );} );}$

2556: ;} ;}$

2557: } }$

2558: $

2559: LENTA\_END

2560: ------>LENTA\_END

Листинг 4 – Разбор исходного кода синтаксическим анализатором

# Приложение Г

|  |
| --- |
| namespace ENV  {  enum ETOKENTYPE { FUN = 0, FIND, IND, SPR, RET, FUNIND, ENVT };  struct EnvironmentToken  {  char lex; //лексема  ETOKENTYPE type; //тип токена  int inLexTable; //вхождение в таблице лексем  int inIdTable; //вхождение в таблице индитификаторов  int parmCount;  int parmLexIndex[TI\_MAX\_COUNT\_PARM];  char id[ID\_MAXSIZE];  EnvironmentToken();  EnvironmentToken(char lex, ETOKENTYPE type, int inLexTable, int inIdTable);  };  struct Environment  {  int level;  char name[ID\_MAXSIZE];  int lexindcount;  int indInIdTable[MAXINDCOUNT]; //массив индексов идентификаторов в таблице  EnvironmentToken indInLexTable[MAXINDCOUNT \* 10];//массив индексов в таблице лексем  int indcount; //количество идентификаторов  int childcount; //количество потомков  Environment\* ParentEnvironment; //ссылка на родитля  Environment\* СhildEnvironments[MAXINDCOUNT]; //массив потомков  void AddInd(int index);  void AddToLex(EnvironmentToken& index);  void SetParent(Environment\* Parent);  void AddChild(Environment\* Child);  void Delete();  void ChangeName(const char\* str);  void Print(std::ofstream& stream);  Environment();  Environment(const char\* name);  };  } |

Листинг 1– Структура окружения

# Приложение Д

|  |
| --- |
| function concat2ss(str1, str2){return str1 + str2;}  function itos1i(number){return String(number);}  function CreateVideo1s(link) {  let iframe = document.createElement("iframe");  iframe.width = "560";  iframe.height = "315";  iframe.src = link;  iframe.allow = "accelerometer; autoplay; clipboard-write; encrypted-media; gyroscope; picture-in-picture; web-share";  iframe.allowFullscreen = true;  return iframe;  }function GetBody0(){return document.body;}  function Create1s(tag){return document.createElement(tag);}  function InsertToHtml2hh(parent, child){parent.appendChild(child);}  function TextSet2hs(tag, text){tag.innerHTML = text;}  function write2hs(tag, text){tag.innerHTML += text;}  function writeln2hs(tag, text){tag.innerHTML += `<br>${text}`;}  function SetCSS2hs(tag, css){tag.style.cssText = css;}  function SetAttribute3hss(tag, atr, value){tag.setAttribute(atr, value);}  function Create2ss( name , css ){  let tag = Create1s( name );  SetCSS2hs( tag , css );  return tag;  }  function CreateButton1s( value ){  let button = Create2ss( 'input' , 'width: 100px; height: 50px; margin: 4px; background-color: gray; border-radius: 10px; ' );  SetAttribute3hss( button , 'type' , 'button' );  SetAttribute3hss( button , 'value' , value );  return button;  }  function main() {  let body = GetBody0( );  let window = Create2ss( 'div' , 'width: 560px; height: 315px; background-color: white; box-sizing: content-box; padding: 5px; border: solid 3px black;' );  InsertToHtml2hh( body , window );  let YoutubeVidoMaker1s = function( shortLink ){  let link = concat2ss( shortLink , '?autoplay=1&loop=1' );  let videoTag = CreateVideo1s( link );  InsertToHtml2hh( body , videoTag );  }  TextSet2hs( window , 'window of number' );  let i = 9;  while( i != 0 ){  let s = itos1i( i );  let button = CreateButton1s( s );  InsertToHtml2hh( body , button );  if( i <= 4 ){  button.addEventListener('click' , () => writeln2hs( window , '' ));  }  button.addEventListener('click' , () => write2hs( window , s ));  i = i - 1;  }  YoutubeVidoMaker1s( 'https://www.youtube.com/embed/X-ANZ8ba8jU' );  }  main(); |

Листинг 1 – Код на языке JavaScript