МИНИСТЕРСТВО ОБРАЗОВАНИЯ РЕСПУБЛИКИ БЕЛАРУСЬ

Учреждение образования «БЕЛОРУССКИЙ ГОСУДАРСТВЕННЫЙ

ТЕХНОЛОГИЧЕСКИЙ УНИВЕРСИТЕТ»

Факультет Информационных Технологий

Кафедра Программной инженерии

Специальность 1-40 01 01 Программное обеспечение информационных технологий

Специализация Программирование интернет-приложений

**ПОЯСНИТЕЛЬНАЯ ЗАПИСКА**

**К КУРСОВОМУ ПРОЕКТУ НА ТЕМУ:**

«Разработка компилятора RMA-2023»

Выполнил студент Русецкий Максим Андреевич

(Ф.И.О.)

Руководитель проекта асс. Мущук Артур Николаевич

(учен. степень, звание, должность, подпись, Ф.И.О.)

Заведующий кафедрой к.т.н. доц. Смеловой Владимир Владиславович

(учен. степень, звание, должность, подпись, Ф.И.О.)

Консультант асс. Мущук Артур Николаевич

(учен. степень, звание, должность, подпись, Ф.И.О.)

Нормоконтролер асс. Мущук Артур Николаевич

(учен. степень, звание, должность, подпись, Ф.И.О.)

Курсовой проект защищен с оценкой

Минск 2023

**Оглавление**

[Введение 4](#_Toc153644851)

[1 Спецификация языка программирования 5](#_Toc153644852)

[1.1 Характеристика языка программирования 5](#_Toc153644853)

[1.2 Определение алфавита языка программирования 5](#_Toc153644854)

[1.3 Применяемые сепараторы 5](#_Toc153644855)

[1.4 Применяемые кодировки 6](#_Toc153644856)

[1.5 Типы данных 6](#_Toc153644857)

[1.6 Преобразование типов данных 7](#_Toc153644858)

[1.7 Идентификаторы 7](#_Toc153644859)

[1.8 Литералы 8](#_Toc153644860)

[1.9 Объявление данных 9](#_Toc153644861)

[1.10 Инициализация данных 10](#_Toc153644862)

[1.11 Инструкции языка 10](#_Toc153644863)

[1.12 Операции языка 11](#_Toc153644864)

[1.13 Выражения и их вычисление 11](#_Toc153644865)

[1.14 Конструкции языка 11](#_Toc153644866)

[1.15 Области видимости идентификаторов 12](#_Toc153644867)

[1.16 Семантические проверки 13](#_Toc153644868)

[1.17 Распределение оперативной памяти на этапе выполнения 13](#_Toc153644869)

[1.18 Стандартная библиотека и её состав 13](#_Toc153644870)

[1.19 Ввод и вывод данных 14](#_Toc153644871)

[1.20 Точка входа 14](#_Toc153644872)

[1.21 Препроцессор 15](#_Toc153644873)

[1.22 Соглашения о вызовах 15](#_Toc153644874)

[1.23 Объектный код 15](#_Toc153644875)

[1.24 Классификация сообщений транслятора 15](#_Toc153644876)

[1.25 Контрольный пример 15](#_Toc153644877)

[2 Структура транслятора 16](#_Toc153644878)

[2.1 Компоненты транслятора, их назначение и принципы взаимодействия 16](#_Toc153644879)

[2.2 Перечень параметров транслятора 17](#_Toc153644880)

[2.3 Протоколы, формируемые транслятором 17](#_Toc153644881)

[3 Разработка лексического анализатора 19](#_Toc153644882)

[3.1 Структура лексического анализатора 19](#_Toc153644883)

[3.2 Входные и выходные данные лексического анализатора 19](#_Toc153644884)

[3.3 Параметры лексического анализатора 19](#_Toc153644885)

[3.4 Алгоритм лексического анализа 19](#_Toc153644887)

[3.5. Контроль входных символов 20](#_Toc153644889)

[3.6 Удаление избыточных символов 21](#_Toc153644890)

[3.7 Перечень ключевых слов 21](#_Toc153644891)

[3.8 Основные структуры данных 23](#_Toc153644892)

[3.9 Структура и перечень сообщений лексического анализатора 25](#_Toc153644893)

[3.10 Принцип обработки ошибок 25](#_Toc153644894)

[3.11 Контрольный пример 25](#_Toc153644895)

[4. Разработка синтаксического анализатора 26](#_Toc153644896)

[4.1 Структура синтаксического анализатора 26](#_Toc153644897)

[4.2 Контекстно-свободная грамматика, описывающая синтаксис языка 26](#_Toc153644898)

[4.3 Построение конечного магазинного автомата 29](#_Toc153644899)

[4.4 Основные структуры данных 30](#_Toc153644900)

[4.5 Описание алгоритма синтаксического разбора 30](#_Toc153644901)

[4.6. Параметры синтаксического анализатора 30](#_Toc153644902)

[4.7 Структура и перечень сообщений синтаксического анализатора 31](#_Toc153644903)

[4.8. Принцип обработки ошибок 31](#_Toc153644904)

[4.9. Контрольный пример 32](#_Toc153644905)

[5 Разработка семантического анализатора 33](#_Toc153644906)

[5.1 Структура семантического анализатора 33](#_Toc153644907)

[5.2 Функции семантического анализатора 33](#_Toc153644908)

[5.3 Структура и перечень сообщений семантического анализатора 33](#_Toc153644909)

[5.4 Принцип обработки ошибок 33](#_Toc153644910)

[5.5 Контрольный пример 34](#_Toc153644911)

[6. Вычисление выражений 35](#_Toc153644912)

[6.1 Выражения, допускаемые языком 35](#_Toc153644913)

[6.2 Польская запись и принцип её построения 35](#_Toc153644914)

[6.3 Программная реализация обработки выражений 35](#_Toc153644915)

[6.4 Контрольный пример 35](#_Toc153644916)

[7. Генерация кода 37](#_Toc153644917)

[7.1 Структура генератора кода 37](#_Toc153644918)

[7.2 Представление типов данных в оперативной памяти 37](#_Toc153644919)

[7.3 Статическая библиотека 37](#_Toc153644920)

[7.4 Особенности алгоритма генерации кода 38](#_Toc153644921)

[7.5 Параметры управляющие генерацией кода 39](#_Toc153644922)

[7.6 Контрольный пример 39](#_Toc153644923)

[8. Тестирование транслятора 40](#_Toc153644924)

[8.1 Общие положения 40](#_Toc153644925)

[8.2 Тестирование проверки на допустимость символов 40](#_Toc153644926)

[8.3 Тестирование лексического анализатора 40](#_Toc153644927)

[8.4 Тестирование синтаксического анализатора 40](#_Toc153644928)

[8.5 Тестирование семантического анализатора 41](#_Toc153644929)

[Заключение 42](#_Toc153644930)

[Список использованных источников 43](#_Toc153644931)

[Приложение А 44](#_Toc153644932)

[Приложение Б 48](#_Toc153644933)

[Приложение В 49](#_Toc153644934)

[Приложение Г 56](#_Toc153644935)

[Приложение Д 68](#_Toc153644936)

**Введение**

Задачей данного курсового проекта была поставлена разработка транслятора своего языка программирования RMA-2023. Он предназначен для выполнения арифметических и логических действий над числами.

Главной задачей транслятора заключается в том, чтобы сделать исходный код на данном языке программирования понятной компьютеру. Для решения этой задачи был выбран способ трансляции исходного в исходный код на языке ассемблера. Язык ассемблера – это машинно-ориентированный язык, представляющий формат записи машинных команд, которые понятны для восприятия человеком.

Исходя из цели курсового проекта, были определены следующие задачи:

* разработка спецификации языка программирования;
* разработка структуры транслятора;
* разработка лексического анализатора;
* разработка синтаксического анализатора;
* разработка семантического анализатора;
* обработка выражений с помощью польской инверсии;
* генерация кода на язык ассемблера;
* тестирование транслятора.

Способы решения каждой задачи будут описаны в соответствующих главах курсового проекта.

В первой главе работы определена спецификация языка программирования.

Во второй главе представлена структура транслятора. В ней перечислены компоненты транслятора, их назначения и принципы взаимодействия.

В третьей главе описана разработка лексического анализатора, который создаёт таблицы лексем и идентификаторов.

В четвертой главе описана разработка синтаксического анализатора, который выполняет разбор исходного кода в соответствии с правилами языка программирования.

В пятой главе описан семантический анализатор, которые проверяет исходный код программы на наличие семантических ошибок.

В шестой главе описан способ преобразования выражений в польский формат.

В седьмой главе представлена генерация кода в язык ассемблера с помощью таблиц лексем и идентификаторов

В восьмой главе описывается тестирование транслятора

**1 Спецификация языка программирования**

**1.1 Характеристика языка программирования**

Язык программирования RMA-2023 является процедурным, строго типизированным, не объектно-ориентированным, компилируемым.

**1.2 Определение алфавита языка программирования**

Алфавит языка программирования – набор символов, которые могут использоваться при написании исходного кода.

При написании программы на языке RMA-2023 используется таблица символов ASCII, представленная на рисунке 1.1.

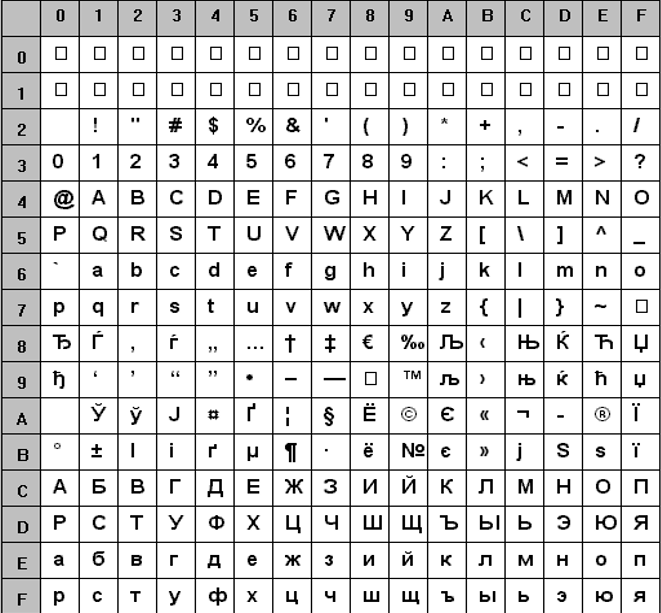


Рисунок 1.1 – Алфавит входных символов

Символы, используемые на этапе выполнения: [a…z], [A…Z], [0…9], символы пробела, табуляции и перевода строки, спецсимволы: [ ] ( ) , ; : # + - / \* > < & !.

**1.3 Применяемые сепараторы**

Символы-сепараторы служат в качестве разделителей цепочек языка во время обработки исходного текста программы с целью разделения на токены. Они представлены в таблице 1.1.

Таблица 1.1 – Символы-сепараторы

|  |  |
| --- | --- |
| Сепаратор | Назначение |
| **=** | Оператор присваивания |

Продолжение таблицы 1.1

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

**1.4 Применяемые кодировки**

Для написания программ язык RMA-2023 использует кодировку ASCII, содержащую английский алфавит, а также некоторые специальные символы, такие как [ ] ( ) , ; : # + - / \* > < & !{}.

**1.5 Типы данных**

В языке RMA-2023 реализованы три типа данных: целочисленный, строковый и символьный. Описание типов приведено в таблице 1.2.

Таблица 1.2 – Типы данных языка RMA-2023

|  |  |
| --- | --- |
| Тип данных | Характеристика |
| Целочисленный тип данных **int** | Фундаментальный тип данных. Используется для работы с числовыми значениями. В памяти занимает 2 байта. Максимальное значение: 32767.  Минимальное значение: -32768.  Инициализация по умолчанию: значение 0.  Поддерживаемые операции:  **+** (бинарный) – оператор сложения;  **++** (унарный) – оператор инкремента;  **--** (унарный) – оператор декремента;  - (бинарный) – оператор вычитания;  **\*** (бинарный) – оператор умножения;  **/** (бинарный) – оператор деления;  **%** (бинарный) – оператор ;  **=** (бинарный) – оператор присваивания. |

Продолжение таблицы 1.2

|  |  |
| --- | --- |
| Тип данных | Характеристика |
| Целочисленный тип данных **int** | В качестве условия цикла/условного оператора поддерживаются следующие логические операции:  **>** (бинарный) – оператор «больше»;  < (бинарный) – оператор «меньше»;  >= (бинарный) – оператор «больше либо равно»;  <= (бинарный) – оператор «меньше либо равно»;  ! (бинарный) – оператор проверки на неравенство.  & (бинарный) – оператор проверки на равенство. |
| Строковый тип данных **string** | Фундаментальный тип данных. Используется для работы с символами, каждый из которых занимает 1 байт. Максимальное количество символов – 255.  Инициализация по умолчанию: строка нулевой длины “”. Операции над данными строкового типа: присваивание строковому идентификатору значения другого строкового идентификатора, строкового литерала или значения строковой функции, а также использование библиотечных функций. |
| Символьный тип данных **symbol** | Фундаментальный тип данных. Используется для работы с одним символом, Инициализация по умолчанию: символ нулевой “”.  Операции над данными символьного типа: присваивание символьному идентификатору значения другого символьного идентификатора, символьного литерала или значения символьной функции. |

**1.6 Преобразование типов данных**

Язык программирования RMA-2023 не обладает средствами для преобразования типов данных

**1.7 Идентификаторы**

Общее количество идентификаторов ограничено максимальным размером таблицы идентификаторов. Идентификаторы должны содержать только символы нижнего регистра латинского алфавита. Максимальная длина идентификатора равна девяти символам.

Идентификаторы, объявленные внутри функционального блока, получают префикс, идентичный имени функции, внутри которой они объявлены. Префикс занимает 8 дополнительных символов. В случае превышения заданной длины, идентификаторы усекаются до длины, равной 17 символов (9 символов на имя идентификатора, 7 символов на префикс). Данные правила действуют для всех типов идентификаторов.

Зарезервированные идентификаторы не предусмотрены. Идентификаторы не должны совпадать с ключевыми словами. Типы идентификаторов: имя переменной, имя функции, параметр функции. Имена идентификаторов-функций не должны совпадать с именами команд ассемблера (это не касается имён идентификаторов-переменных). Примеры идентификаторов приведены в таблице 1.3.

Таблица 1.3 – Пример идентификаторов

|  |  |
| --- | --- |
| Идентификатор | Пример |
| Корректные | Int1  x |
| Некорректные | 1do  int |

**1.8 Литералы**

С помощью литералов осуществляется инициализация переменных. Все литералы являются **rvalue**. Типы литералов языка RMA-2023 представлены в таблице 1.4.

Таблица 1.4 – Описание литералов

|  |  |  |
| --- | --- | --- |
| Литералы | Пояснение | Пример |
| Целочисленные литералы в десятичном представлении | Последовательность цифр 0…9 с предшествующим знаком минус или без него | type int x;  x = 12;  12 – целочисленный литерал в десятичном представлении. |
| Целочисленные литералы в восьмеричном представлении | Последовательность цифр 0…7 с предшествующим символом ‘0’ и предшествующим знаком минус или без него | type int y;  y = 0120;  0120 –  целочисленный литерал в восьмеричном представлении. |
| Целочисленные литералы в двоичном представлении | Последовательность цифр 0,1 с предшествующими символами ‘0b’ и предшествующим знаком минус или без него | type int z;  y = 0b1010;  0b1010 –  целочисленный литерал в двоичном представлении |

Продолжение таблицы 1.4

|  |  |  |
| --- | --- | --- |
| Литералы | Пояснение | Пример |
| Целочисленные литералы в шестнадцатеричном представлении | Последовательность цифр 0..9 и символов A..F с предшествующими символами ‘0x’ и предшествующим знаком минус или без него | type int h;  y = 0x1010;  0x1010 –  целочисленный литерал в  шестнадцатеричном представлении |
| Строковые литералы | Набор символов (от 1 до 252), заключённых в двойные кавычки | type string str  str = “Hello”;  “Hello” – строковый литерал. |
| Символьные литералы | Отдельный символ,  заключённых в одинарные кавычки | type symbol s;  s = ‘S’;  ‘S’ – символьный литерал. |
| Логические литералы | Число или бинарная операция с использованием логического оператора | type int a = 5;  if a > 4  a > 4 – логический литерал. |

Ограничения на строковые и символьные литералы языка RMA-2023: внутри литерала не допускается использование символов кириллицы, а также одинарных и двойных кавычек, логические литералы могут использоваться только в условии **if** или **while**.

**1.9 Объявление данных**

Для объявления переменной используется ключевое слово **type**, после которого указывается тип данных и имя идентификатора. Допускается инициализация при объявлении.

Пример объявления числового типа с инициализацией:

* type int int1 = 12;
* type int int2 = 0x1F;

Пример объявления переменной строкового типа с инициализацией:

* type string str= “Hello”;

Пример объявления переменной символьного типа с инициализацией:

* type symbol s= “S”;

Для объявления функций используется ключевое слово **function**, перед которым указывается тип функции (если функция возвращает значение), или ключевое слово **void**, если функция ничего не возвращает, а после – имя функции либо процедуры. Далее обязателен список параметров и тело функции. Точкой входа в программу является функция **main**, которая объявляется без указания типа возвращаемого значения и ключевого слова **function**.

Область видимости идентификаторов «сверху вниз» (по принципу С++). В языке RMA-2023 требуется обязательное объявление переменной перед её использованием. Все переменные должны находиться внутри программного блока языка. Имеется возможность объявления одинаковых переменных в разных блоках.

**1.10 Инициализация данных**

При объявлении переменной допускается инициализация данных. При этом переменной будет присвоено значение литерала или идентификатора, стоящего справа от знака равенства. Объектами-инициализаторами могут быть только идентификаторы или литералы. При объявлении без инициализации предусмотрены значения по умолчанию: значение 0 для типа **int**, строка длины 0 (“”) для типа **string** и пустой символ (‘’) для типа **symbol**.

**1.11 Инструкции языка**

Инструкции языка RMA-2023 представлены в таблице 1.5.

Таблица 1.5 – ИнструкцииязыкаRMA-2023

|  |  |
| --- | --- |
| Инструкция | Реализация |
| Объявление переменной | type <тип данных> <идентификатор>; |
| Объявление переменной с явной инициализацией | type <тип данных> <идентификатор> =  <идентификатор/литерал>; |
| Возврат из функции или процедуры | Для функций, возвращающих значение:  return <идентификатор/литерал>;  Для процедур:  return; |
| Вывод данных | write <идентификатор/литерал>;  writeline <идентификатор/литерал>; |
| Вызов функции или процедуры | <идентификатор функции> (<список параметров>);  Список параметров может быть пустым. |
| Условный оператор | if <условное выражение>  istrue[…]  isfalse[…] |
| Оператор цикла | while <условное выражение>  do[…] |
| Присваивание | <идентификатор> = <выражение>;  Выражением может быть идентификатор, литерал, или вызов функции соответствующего типа. Для целочисленного типа выражение может быть дополнено арифметическими операциями с любым количеством операндов с использованием скобок. |

**1.12 Операции языка**

В языке RMA-2023 предусмотрены следующие операции с данными. Приоритетность операции умножения выше приоритета операций сложения и вычитания и операции инкремента и декремента. Для установки наивысшего приоритета используются круглые скобки. Условные операции представлены шестью операторами языка. Операции языка представлены в таблице 1.6

Таблица 1.6 – Операции языка RMA-2023

|  |  |
| --- | --- |
| Тип оператора | Оператор |
| Арифметические | 1. **+** –сложение 2. **-** – вычитание 3. \* – умножение 4. **/** – деление 5. = – присваивание 6. ++ – инкремент 7. -- – декремент |
| Строковые | 1. **=** – присваивание |
| Логические | 1. > – больше  2. < – меньше  3. >= – больше либо равно  4. <= – меньше либо равно  5. ! – проверка на неравенство  6. & – проверка на равенство |
| Символьные | 1**.** = – присваивание |

**1.13 Выражения и их вычисление**

Вычисление выражений – одна из важнейших задач языков программирования. Всякое выражение составляется согласно следующим правилам:

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

Перед генерацией кода каждое выражение приводится к записи в польской записи для удобства дальнейшего вычисления выражения на языке ассемблера.

**1.14 Конструкции языка**

Программа на языке RMA-2023 оформляется в виде функций пользователя и главной функции. При составлении функций рекомендуется выделять блоки и фрагменты и применять отступы для лучшей читаемости кода.

Ключевые программные конструкции языка программирования RMA-2023 представлены в таблице 1.7.

Таблица 1.7 – Программные конструкции языка RMA-2023

|  |  |
| --- | --- |
| Конструкция | Реализация |
| Главная функция | main{… } |
| Внешняя функция | <тип данных> function <идентификатор> (<тип> <идентификатор>, ...)  {…  return <идентификатор/литерал>;  } |
| Внешняя процедура | void function <идентификатор> ( <тип> <идентификатор>, ...)  {…  return;  } |
| Условная конструкция | if <условное выражение>  istrue[…]  isfalse[…]  <условное выражение> - может представлять собой два идентификатора или один идентификатор и один литерал, а также условный оператор между ними, или один идентификатор или литерал (в таком случае идентификатор или литерал используется в качестве булевого значения, то есть условие истинно, если значение не равно 0) |
| Цикл | while <условное выражение>  do[…]  <условное выражение> - может представлять собой два идентификатора или один идентификатор и один литерал, а также условный оператор между ними, или один идентификатор или литерал (в таком случае идентификатор или литерал используется в качестве булевого значения, то есть условие истинно, если значение не равно 0) |

**1.15 Области видимости идентификаторов**

Область видимости: сверху вниз (как и в С++). Переменные, объявленные в одной функции, недоступны в другой.

Все объявления и операции с переменными происходят внутри какого-либо блока. Каждая переменная или параметр функции получают префикс – название функции, внутри которой они находятся.

Все идентификаторы являются локальными и обязаны быть объявленными внутри какой-либо функции. Глобальных переменных нет. Параметры видны только внутри функции, в которой объявлены.

**1.16 Семантические проверки**

В языке программирования RMA-2023 выполняются следующие семантические проверки:

* наличие точки входа в программу;
* наличие типа идентификатора;
* совпадение типов данных в выражении;
* деление на ноль;
* проверка типа возвращаемого значения;
* проверка передаваемых параметров;
* единственность точки входа;
* переопределение идентификаторов;
* использование идентификаторов без их объявления;
* правильность строковых и символьных выражений;
* превышение размера символьных, строковых и числовых литералов;
* правильность составленного условия цикла/условного оператора.

**1.17 Распределение оперативной памяти на этапе выполнения**

Транслированный код использует две области памяти. В сегмент констант заносятся все литералы. В сегмент данных заносятся переменные и параметры функций. Локальная область видимости в исходном коде определяется за счет использования правил именования идентификаторов и регулируется их префиксами, что и обуславливает их локальность на уровне исходного кода, несмотря на то, что в оттранслированным в язык ассемблера коде переменные имеют глобальную область видимости.

**1.18 Стандартная библиотека и её состав**

В языке RMA-2023 присутствует стандартная библиотека, которая подключается автоматически при трансляции исходного кода в язык ассемблера. Содержимое библиотеки и описание функций представлено в таблице 1.8.

Таблица 1.8 – Стандартная библиотека языка RMA-2023

|  |  |  |  |
| --- | --- | --- | --- |
| Имя функции | Возвращаемое значение | Принимаемые параметры | Описание |
| length | int | string str – строка | Вычисление длины строки |
| random | int |  | Возвращение случайного числа |

Стандартная библиотека написана на языке С++, подключается к транслированному коду на этапе генерации кода. Вызовы стандартных функций доступны там же, где и вызов пользовательских функций. Также в стандартной библиотеке реализованы функции для манипулирования выводом, недоступные конечному пользователю. Для вывода предусмотрен операторы **write** и **writeline**. Эти функции представлены в таблице 1.9.

Таблица 1.9 – Стандартная библиотека языка RMA-2023

|  |  |  |  |
| --- | --- | --- | --- |
| Имя функции | Возвращаемое значение | Принимаемые параметры | Описание |
| outnum | int | int value – число | Функция для вывода в стандартный поток значения целочисленного идентификатора/литерала. |
| outnumline | Int | int value – число | Функция для вывода в стандартный поток значения целочисленного идентификатора/литерала и переводит на новую строку |
| outstr | int | char\* ptr – массив символов | Функция для вывода в стандартный поток значения строкового или символьного  идентификатора/литерала. |
| outstrline | Int | char\* ptr – массив символов | Функция для вывода в стандартный поток значения строкового или символьного  идентификатора/литерала.  И переводит на новую строку |
| system\_pause | int |  | Функция для остановки программы |

**1.19 Ввод и вывод данных**

Вывод данных осуществляется с помощью операторов **write, writeline**. Допускается использование операторов **write, writeline** с литералами и идентификаторами.

Функции, управляющие выводом данных, реализованы на языке С++ и вызываются из транслированного кода, конечному пользователю недоступны. Пользовательская команда **write, writeline** в транслированном коде будут заменена вызовом нужных библиотечных функций. Библиотека, содержащая нужные процедуры, подключается на этапе генерации кода.

**1.20 Точка входа**

В языке RMA-2023 каждая программа должна содержать главную функцию (точку входа) **main**, с первой инструкции которой начнётся последовательное выполнение команд программы. Главная функция **main** не имеет типа возвращаемого значения, принимаемых параметров.

**1.21 Препроцессор**

Препроцессор, принимающий и выдающий некоторые данные на вход транслятору, в языке RMA-2023 отсутствует.

**1.22 Соглашения о вызовах**

В языке вызов функций происходит по соглашению о вызовах stdcall. Особенности stdcall:

* все параметры функции передаются через стек;
* память высвобождает вызываемый код;
* занесение в стек параметров идёт справа налево.

**1.23 Объектный код**

Язык RMA-2023 транслируется в язык ассемблера, а затем - в объектный код.

**1.24 Классификация сообщений транслятора**

Генерируемые транслятором сообщения определяют степень его информативности, то есть сообщения транслятора должны давать максимально полную информацию о допущенной пользователем ошибке при написании программы. Сообщения транслятора приведены в таблице 1.10, а также в приложении А.

Таблица 1.10 – Классификация ошибок

|  |  |
| --- | --- |
| Номера ошибок | Характеристика |
| 0 – 200 | Системные ошибки |
| 200 – 299 | Ошибки лексического анализа |
| 300 – 399 | Ошибки семантического анализа |
| 600 – 699 | Ошибки синтаксического анализа |
| 400-499, 700-999 | Зарезервированные коды ошибок |

**1.25 Контрольный пример**

Контрольный пример демонстрирует главные особенности языка RMA-2023: его фундаментальные типы, основные структуры, функции, процедуры, использование функций статической библиотеки. Исходный код контрольного примера представлен в приложении А.

**2 Структура транслятора**

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

В языке RMA-2023 исходный код транслируется в язык Assembler. Транслятор языка разделён на отдельные части, которые взаимодействуют между собой и выполняют отведённые им функции. Для того чтобы получить ассемблерный код, используется выходные данные работы лексического анализатора, а именно таблица лексем и таблица идентификаторов. Для указания выходных файлов используются входные параметры транслятора, которые описаны в таблице 2.1. Структура транслятора языка RMA-2023 приведена на рисунке 2.1.



Рисунок 2.1 – Структура транслятора языка программирования RMA-2023

Первая стадия работы компилятора называется лексическим анализом, а программа, её реализующая, – лексическим анализатором (сканером). На вход лексического анализатора подаётся последовательность символов входного языка. Он производит предварительный разбор текста, преобразующий единый массив текстовых символов в массив отдельных слов (в теории компиляции вместо термина «слово» часто используют термин «токен»). 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. Таблица лексем (ТЛ) и таблица идентификаторов (ТИ) являются входом для следующей фазы компилятора – синтаксического анализа (разбора, парсера).

Цели лексического анализатора:

* убрать все лишние пробелы и комментарии;
* выполнить распознавание лексем;
* построить таблицу лексем и таблицу идентификаторов;
* при неуспешном распознавании или обнаружении некоторых ошибок во входном тексте выдать сообщение об ошибке.

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

Семантический анализатор – часть транслятора, выполняющая семантический анализ, то есть проверку исходного кода на наличие ошибок, которые невозможно отследить при помощи регулярной и контекстно-свободной грамматики. Входными данными являются таблица лексем и идентификаторов.

Генератор кода – часть транслятора, выполняющая генерацию ассемблерного кода на основе полученных данных на предыдущих этапах трансляции. На вход генератора подаются таблица лексем и таблица идентификаторов, на основе которых генерируется файл с ассемблерным кодом.

**2.2 Перечень 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 RMA-2023

|  |  |  |
| --- | --- | --- |
| Входной параметр | Описание параметра | Значение по умолчанию |
| -in:<путь к in-файлу> | Файл с исходным кодом на языке RMA-2023 , 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> | Выходной файл с кодом на языке ассемблера, имеющий расширение .asm | Значение по умолчанию:  <имя in-файла>.asm |

**2.3 Протоколы, формируемые транслятором**

В ходе работы программы формируются протоколы работы лексического, синтаксического и семантического анализаторов, которые содержат в себе перечень протоколов работы. В таблице 2.2 приведены протоколы, формируемые транслятором и их содержимое.

Таблица 2.2 – Протоколы, формируемые транслятором языка RMA-2023

|  |  |
| --- | --- |
| Файл журнала, заданный параметром "-log:" | Файл с протоколом работы транслятора языка программирования RMA-2023 . Содержит таблицу лексем и таблицу идентификаторов, протокол работы синтаксического анализатора и дерево разбора, полученные на этапе лексического и синтаксического анализа, а также результат работы алгоритма преобразования выражений к польской записи. |
| Выходной файл, c расширением ".asm" | Результат работы программы – файл, содержащий исходный код на языке ассемблера. |

**3 Разработка лексического анализатора**

**3.1 Структура лексического анализатора**

Лексический анализатор – часть транслятора, выполняющая лексический анализ. Лексический анализатор принимает обработанный и разбитый на отдельные компоненты исходный код на языке RMA-2023. На выходе формируется таблица лексем и таблица идентификаторов. Структура лексического анализатора представлена на рисунке 3.1.



Рисунок 3.1 – Структура лексического анализатора

**3.2 Входные и выходные данные лексического анализатора**

Входными данными лексического анализатора является обработанный и разбитый на отдельные токены исходный код на языке RMA-2023. Результатом работы лексического анализатора становятся таблица лексем и таблица идентификаторов, которые выводятся как в файл журнала, так и в командную строку.

**3.3 Параметры лексического анализатора**

Язык программирования RMA-2023 не имеет дополнительных параметров, влияющих на работу лексического анализатора.

**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 «string» представлен на рисунке 3.2, где S0 – начальное, а S6 – конечное состояние автомата.

s t r i n g

Рисунок 3.2 – Пример графа переходов для цепочки string

**3.5. Контроль входных символов**

Для оптимизации и удобной работы с исходным кодом, при передаче его в лексический анализатор, все символы разделяются по категориям. Таблица входных символов представлена на рисунке 3.3, категории входных символов представлены в таблице 3.1.

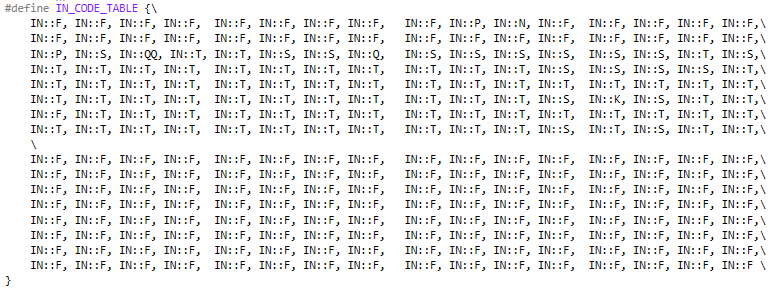


Рисунок 3.3. – Таблица контроля входных символов

Таблица 3.1 – Соответствие символов и их значений в таблице

|  |  |
| --- | --- |
| Значение в таблице входных символов | Символы |
| Разрешенный | T |
| Запрещенный | F |
| Игнорируемый | I |

Продолжение таблицы 3.1

|  |  |
| --- | --- |
| Значение в таблице входных символов | Символы |
| Одинарная кавычка | Q |
| Двойная кавычка | QQ |
| Сепаратор | S |
| Перевод строки | N |
| Пробел, табуляция | P |
| Обратный слеш | K |

**3.6 Удаление избыточных символов**

Избыточными символами являются символы табуляции, пробелы и комментарии. Избыточные символы удаляются на этапе разбиения исходного кода на токены.

Описание алгоритма удаления избыточных символов:

* Посимвольно считываем файл с исходным кодом программы;
* Встреча пробела или знака табуляции является своего рода встречей символа-сепаратора;
* В отличие от других символов-сепараторов не записываем в очередь лексем эти символы, т.е. игнорируем.

**3.7 Перечень ключевых слов**

Лексический анализатор преобразует исходный текст, заменяя лексические единицы лексемами для создания промежуточного представления исходной программы. Соответствие токенов и лексем приведено в таблице 3.2.

Таблица 3.2 – Соответствие токенов и сепараторов с лексемами

|  |  |  |
| --- | --- | --- |
| Токен | Лексема | Пояснение |
| int, string,  symbol | t | Названия типов данных языка. |
| Идентификатор | i | Длина идентификатора – 9 символов. |
| Литерал | l | Литерал любого доступного типа. |
| void | g | Ключевое слово для процедур – функций, не возвращающих значения. Указывается перед словом function. |
| return | e | Выход из функции/процедуры. |
| main | m | Главная функция. |
| type | n | Объявление переменной. |
| write | o | Вывод данных. |
| writeline | ^ | Вывод строки с данными. |
| if | ? | Условная конструкция. |

Продолжение таблицы 3.2

|  |  |  |
| --- | --- | --- |
| Токен | Лексема | Пояснение |
| istrue | r | Истинная ветвь условного оператора. |
| isfalse | w | Ложная ветвь условного оператора. |
| do | d | Указывает на начало тела цикла. |
| while | c | Условие выполнения цикла |
| ; | ; | Разделение выражений. |
| , | , | Разделение параметров функций. |
| [ | [ | Начало блока/тела функции. |
| ] | ] | Закрытие блока/тела функции. |
| { | { | Начало области видимости |
| } | } | Конец области видимости |
| ( | ( | Передача параметров в функцию, приоритет операций. |
| ) | ) | Закрытие блока для передачи параметров, приоритет операций. |
| = | = | Знак присваивания. |
| +  -  \*  /  ++  -- | +  -  \*  /  :  : | Знаки операций. |
| >  <  !  &  >=  <= | >  <  !  &  x  z | Знаки логических операторов |

Пример реализации таблицы лексем представлен в приложении Б.

Каждому выражению соответствует детерминированный конечный автомат, по которому происходит разбор данного выражения. На каждый автомат в массиве подаётся токен и с помощью регулярного выражения, соответствующего данному графу переходов, происходит разбор. 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 и пример графа перехода конечного автомата показаны в листингах 3.3 и 3.4 соответственно.

|  |
| --- |
| struct RELATION {  unsigned char symbol;  short nnode;  RELATION(  unsigned char c = 0x00,  short ns = 0  );  };  struct NODE {  short n\_relation;  RELATION\* relations;  NODE();  NODE  (short n,  RELATION rel, ...  );  };  struct FST{  unsigned char\* string;  short position;  short nstates;  NODE\* nodes;  short\* rstates;  FST(short ns,  NODE n, ...);  FST(unsigned char\* s,  FST& fst);    };  bool execute(  FST& fst);  } |

Листинг 3.3 – Структура конечного автомата

|  |
| --- |
| #define GRAPH\_SYMBOL 7, \  FST::NODE(1, FST::RELATION('s',1)),\  FST::NODE(1, FST::RELATION('y',2)),\  FST::NODE(1, FST::RELATION('m',3)),\  FST::NODE(1, FST::RELATION('b',4)),\  FST::NODE(1, FST::RELATION('o',5)),\  FST::NODE(1, FST::RELATION('l',6)),\  FST::NODE() |

Листинг 3.4 – Пример реализации графа конечного автомата для токена symbol

**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), позицию в исходном коде (pos) и номер в таблице идентификаторов, если лексема является идентификаторов (idxTI). Таблица идентификаторов содержит имя идентификатора (id), номер в таблице лексем (idxfirstLE), тип данных (iddatatype), тип идентификатора (idtype) и значение (или параметры функций) (value).

Код C++ со структурой таблицы лексем представлен на рисунке 3.3. Код C++ со структурой таблицы идентификаторов представлен на рисунке 3.4.

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

Листинг 3.5 – Структура таблицы лексем

|  |
| --- |
| struct Entry{  int idxfirstLE;  char id[SCOPED\_ID\_MAXSIZE];  IDDATATYPE iddatatype;  IDTYPE idtype;    union  {  int vint;  struct  {  int len;  char str[STR\_MAXSIZE - 1];  } vstr;  char symbol;  struct  {  int count;  IDDATATYPE\* types;  } params;  } value |

Листинг 3.6 – Структура таблицы идентификаторов

**3.9 Структура и перечень сообщений лексического анализатора**

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. Если в процессе лексического анализа будет выявлена ошибка, работа компилятора будет закончена. Перечень сообщений представлен в листинге 3.7.

|  |
| --- |
| ERROR\_ENTRY(100, "Системная ошибка: Параметр -in должен быть задан"),  ERROR\_ENTRY(101, "Системная ошибка: Превышена длина входного параметра"),  ERROR\_ENTRY(102, "Системная ошибка: Ошибка при открытии файла с исходным кодом(-in)"),  ERROR\_ENTRY(103, "Системная ошибка: Ошибка при создании файла протокола(-log)"), |

Листинг 3.7 – Сообщения лексического анализатора

**3.10 Принцип обработки ошибок**

Ошибки, возникающие в процессе трансляции программы, фиксируются в протокол, заданный входным параметрами. В случае возникновения ошибок происходит их протоколирование с номером ошибки и диагностическим сообщением.

**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.

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

В синтаксическом анализаторе транслятора языка RMA-2023 используется контекстно-свободная грамматика типа II в иерархии Хомского (Контекстно-свободная грамматика) , где:

* T – множество терминальных символов (было описано в разделе 1.2 данной пояснительной записки)
* N – множество нетерминальных символов (первый столбец таблицы 4.1)
* P – множество правил языка (второй столбец таблицы 4.1)
* 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:

* , где ; (или , или );
* , 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

|  |  |  |
| --- | --- | --- |
| Символ | Правила | Какие правила порождает |
| S | S->tfiPTS  S->pfiPGS  S->m{K}  S->m{} | Стартовые правила, описывающее общую структуру программы |
| P | P->(E)  P->() | Правила для параметров объявляемых функций |
| T | T->{Q}  T->{KQ} | Правила для тела функций |
| G | G->{e;}  G->{Ke;} | Правила для тела процедур |
| E | E->ti,E  E->ti | Правила для списка параметров функции |
| F | F->(N)  F->() | Правила для вывозов функций(в т.ч. и в выражениях) |
| N | N->i  N>l  N->l,N  N->I,N | Правила для параметров вызываемых функций |
| R | R->rY  R>wY  R->rYwY  R->wYrY | Правила составления цикла/условного оператора |
| Z | Z->iLi  Z->iLl  Z->lLl  Z->lLi  Z->i  Z->l | Правила для условия цикла/условного оператора |
| L | L-><  L->>  L->&  L->!  L->x  L->z | Правила для логических операторов |

Продолжение таблицы 4.1

|  |  |  |
| --- | --- | --- |
| Символ | Правила | Какие правила порождает |
| V | V->l  V->i | Правила для простых выражений |
| A | A->+  A->-  A->\*  A->%  A->/ | Правила для арифметических и свдиговых операторов |
| Y | Y->[XQ]  Y->[X]  Y->[Q] | Правила для тела цикла/условного выражения |
| W | W->l  W->i  W->(W)  W->(-i)  W->(-i)AW  W->(W)AW  W->iF  W->pF  W->iAW  W->lAW  W->iFAW  W->pFAW | Правила для сложных выражений |
| K | K->nti=V;K  K->nti=W;K  K->nti;K  K->i=W;K  K->i:;K  K->oV;K  K->^V;K  K->?ZRK  K->cZdHK  K->i=UF;K  K->iF;K  K->nti=V;  K->nti=W;  K->i=W;  K->i:;  K->nti;  K->oV;  K->^V;  K->?ZR  K->cZdH | Программные конструкции |

Продолжение таблицы 4.1

|  |  |  |
| --- | --- | --- |
| K | K->i=UF;  K->iF; |  |
| X | X->i=W;X  X->i:;X  X->oV;X  X->^V;X  X->i=UF;K  X->iF;K  X->i=W;  X->oV;  X->^V;  X->i:;  X->i=UF;  X->iF; | Программные конструкции внутри цикла/условного оператора |
| Q | Q->eV;  Q->e(-i);  Q->e;  Q->e(l);  Q->eW; | Программная конструкция return |
| U | U->i  U->p | Правила для идентификатора функции |
| H | H->[X] | Конструирование тела цикла |

**4.3 Построение конечного магазинного автомата**

Конечный автомат с магазинной памятью представляет собой семерку, описание которой 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 $) |

Продолжение таблицы 4.2

|  |  |  |
| --- | --- | --- |
|  | Функция переходов автомата | Функция представляет из себя множество правил грамматики, описанных в таблице 4.1. |
|  | 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 |

**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 RMA-2023. Данные структуры представлены в приложении В.

**4.5 Описание алгоритма синтаксического разбора**

Принцип работы синтаксического анализатора, следующий:

1. В магазин записывается стартовый символ;
2. На основе полученных ранее таблиц формируется входная лента;
3. Запускается автомат;
4. Выбирается цепочка, соответствующая нетерминальному символу, записывается в магазин в обратном порядке;
5. Если терминалы в стеке и в ленте совпадают, то данный терминал удаляется из ленты и стека. Иначе возвращаемся в предыдущее сохраненное состояние и выбираем другую цепочку нетерминала;
6. Если в магазине встретился нетерминал, переходим к пункту 4;
7. Если наш символ достиг дна стека, и лента в этот момент пуста, то синтаксический анализ выполнен успешно. Иначе генерируется исключение.

**4.6. Параметры синтаксического анализатора**

Входной информацией для синтаксического анализатора является таблица лексем и идентификаторов.

Кроме того, используется описание грамматики в форме Грейбах. Результаты работы лексического разбора, а именно дерево разбора и протокол работы автомата с магазинной памятью выводятся в журнал работы программы.

**4.7 Структура и перечень сообщений синтаксического анализатора**

Перечень сообщений синтаксического анализатора представлен на рисунке – 4.2.

|  |
| --- |
| ERROR\_ENTRY(600, "Синтаксическая ошибка: Неверная структура программы"),  ERROR\_ENTRY(601, "Синтаксическая ошибка: Не найден список параметров функции"),  ERROR\_ENTRY(602, "Синтаксическая ошибка: Ошибка в теле функции"),  ERROR\_ENTRY(603, "Синтаксическая ошибка: Ошибка в списке параметров функции"),  ERROR\_ENTRY(604, "Синтаксическая ошибка: Ошибка в вызове функции/выражении"),  ERROR\_ENTRY(605, "Синтаксическая ошибка: Ошибка в списке фактических параметров функции"),  ERROR\_ENTRY(606, "Синтаксическая ошибка: Ошибка при констуировании условного выражения"),  ERROR\_ENTRY(607, "Синтаксическая ошибка: Ошибка при конструировании return"),  ERROR\_ENTRY(608, "Синтаксическая ошибка: Ошибка в условии цикла/условного выражения"),  ERROR\_ENTRY(609, "Синтаксическая ошибка: Неверный условный оператор"),  ERROR\_ENTRY(610, "Синтаксическая ошибка: Неверный арифметический оператор"),  ERROR\_ENTRY(611, "Синтаксическая ошибка: Ошибка в арифметическом выражении"),  ERROR\_ENTRY(612, "Синтаксическая ошибка: Недопустимая синтаксическая конструкция"),  ERROR\_ENTRY(613, "Синтаксическая ошибка: Недопустимая синтаксическая конструкция в теле цикла/условного выражения"),  ERROR\_ENTRY(614, "Синтаксическая ошибка: Ошибка при конструировании условного выражения в цикле"),  ERROR\_ENTRY(615, "Синтаксическая ошибка: Ожидался литерал или идентификатор"),  ERROR\_ENTRY(616, "Синтаксическая ошибка: Ошибка в теле условного выражения"), |

Листинг 4.2 – Сообщения синтаксического анализатора

**4.8. Принцип обработки ошибок**

Синтаксический анализатор выполняет разбор исходной последовательности лексем до тех пор, пока не дойдёт до конца цепочки лексем. При ошибке анализ останавливается и выводится сообщение.

**4.9. Контрольный пример**

Результаты работы лексического разбора, а именно дерево разбора и протокол работы автомата с магазинной памятью приведены в приложении В. Дерево разбора исходного кода также представлено в приложении В.

**5 Разработка семантического анализатора**

**5.1 Структура семантического анализатора**

Семантический анализатор принимает на свой вход результаты работ лексического и синтаксического анализаторов, то есть таблицы лексем, идентификаторов и результат работы синтаксического анализатора, то есть дерево разбора, и последовательно ищет необходимые ошибки. Некоторые проверки (такие как проверка на единственность точки входа, проверка на предварительное объявление переменной) осуществляются в процессе лексического анализа. Общая структура обособленно работающего (не параллельно с лексическим анализом) семантического анализатора представлена на рисунке 5.1.



Рисунок 5.1. Структура семантического анализатора

**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, формируемые семантическим анализатором, представлены в приложении Б.

**5.4 Принцип обработки ошибок**

Ошибки, возникающие в процессе трансляции программы, фиксируются в протокол, заданный входным параметрами.

В случае возникновения ошибок происходит их протоколирование с номером ошибки и диагностическим сообщением. Анализ останавливается после того, как будут найдены все 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{  int a = 5;  } | Ошибка N304: Семантическая ошибка: В объявлении отсутствует ключевое слово type Строка: 2 Позиция в строке: 1 |
| main{  type int a = 5;  type string b = a;  } | Ошибка N314: Семантическая ошибка: Типы данных в выражении не совпадают Строка: 3 Позиция в строке: 17 |

**6. Вычисление выражений**

**6.1 Выражения, допускаемые языком**

В языке RMA-2023 приоритет операций представлен на таблице 6.1.

Таблица 6.1 – Приоритеты операций

|  |  |
| --- | --- |
| Операция | Значение приоритета |
| ( ) | 0 |
| + | 2 |
| - | 2 |
| \* | 3 |
| / | 3 |
| % | 3 |

**6.2 Польская запись и принцип её построения**

Все выражения языка RMA-2023 преобразовываются к обратной польской записи. Польская запись — это альтернативный способ записи арифметических выражений, преимущество которого состоит в отсутствии скобок. Существует два типа польской записи: прямая и обратная, также известные как префиксная и постфиксная. Отличие их от классического, инфиксного способа заключается в том, что знаки операций пишутся не между, а, соответственно, до или после аргументов.

Алгоритм построения польской записи:

* исходная строка: выражение;
* результирующая строка: польская запись;
* стек: пустой;
* исходная строка просматривается слева направо;
* операнды переносятся в результирующую строку;
* операция записывается в стек, если стек пуст;
* операция выталкивает все операции с большим или равным приоритетом в результирующую строку;
* отрывающая скобка помещается в стек;
* закрывающая скобка выталкивает все операции до открывающей скобки, после чего обе скобки уничтожаются.

**6.3 Программная реализация обработки выражений**

Программная реализация алгоритма преобразования выражений к польской записи представлена в приложении Г.

**6.4 Контрольный пример**

Пример преобразования выражений из контрольных примеров к обратной польской записи представлен в таблице 6.2.

Преобразование выражений в формат польской записи необходимо для построения более простых алгоритмов их вычисления и преобразования к ассемблерному коду. В приложении Г приведены изменённые таблицы лексем и идентификаторов, отображающие результаты преобразования выражений в польский формат.

Таблица 6.2 – Преобразование выражений к ПОЛИЗ

|  |  |
| --- | --- |
| Выражение | Обратная польская запись для выражения |
| i[2]=(((l[3]+l[4])-i[0])\*l[5])/l[6]; | i[2]=l[3]l[4]+i[0]-l[5]\*l[6]/ |
| i[23]=(i[23]+l[26])\*l[26] | i[23]=i[23]l[26]+l[26]\* |
| i[3]=(((l[4]+l[5])-i[0])\*l[6]) | i[3]=l[4]l[5]+i[0]-l[6]\* |

**7. Генерация кода**

**7.1 Структура генератора кода**

В языке RMA-2023 генерация кода является заключительным этапом трансляции. Генератор принимает на вход таблицы лексем и идентификаторов, полученные в результате лексического анализа. В соответствии с таблицей лексем строится выходной файл на языке ассемблера, который будет являться результатом работы транслятора. В случае возникновения ошибок генерация кода не будет осуществляться. Структура генератора кода RMA-2023 представлена на рисунке 7.1.



Рисунок 7.1 – Структура генератора кода

**7.2 Представление типов данных в оперативной памяти**

Элементы таблицы идентификаторов расположены сегментах .data и .const языка ассемблера. Соответствия между типами данных идентификаторов на языке RMA-2023 и на языке ассемблера приведены в таблице 7.1.

Таблица 7.1 – Соответствия типов идентификаторов языка RMA-2023 и языка ассемблера

|  |  |  |
| --- | --- | --- |
| Тип идентификатора на языке RMA-2023 | Тип идентификатора на языке ассемблера | Пояснение |
| int | sword | Хранит целочисленный тип данных. |
| string | word | Хранит указатель на начало строки. Строка должна завешаться нулевым символом. |
| symbol | word | Хранит символьный тип данных |

**7.3 Статическая библиотека**

В языке RMA-2023 предусмотрена статическая библиотека. Статическая библиотека содержит функции, написанные на языке C++.

Объявление функций статической библиотеки генерируется автоматически в коде ассемблера. Объявление функций статической библиотеки генерируется автоматически.

Таблица 7.3 – Функции статической библиотеки

|  |  |
| --- | --- |
| Функция | Назначение |
| int outnum(int num) | Функция для вывода в стандартный поток значения целочисленного идентификатора/литерала. |
| int outnumline(int num) | Функция для вывода в стандартный поток значения целочисленного идентификатора/литерала и переводит на новую строку |
| int outstr(char\* str) | Функция для вывода в стандартный поток значения строкового или символьного  идентификатора/литерала. |
| int outstrline(char str\*) | Функция для вывода в стандартный поток значения строкового или символьного  идентификатора/литерала.  И переводит на новую строку |
| int system\_pause() | Функция для остановки программы |
| int length(char\* str) | Функция подсчета длины строки |
| int random() | Функция генерирования случайного числа |

**7.4 Особенности алгоритма генерации кода**

В языке RMA-2023 генерация кода строится на основе таблиц лексем и идентификаторов. Схема работы генератора кода представлена на рисунке 7.2.

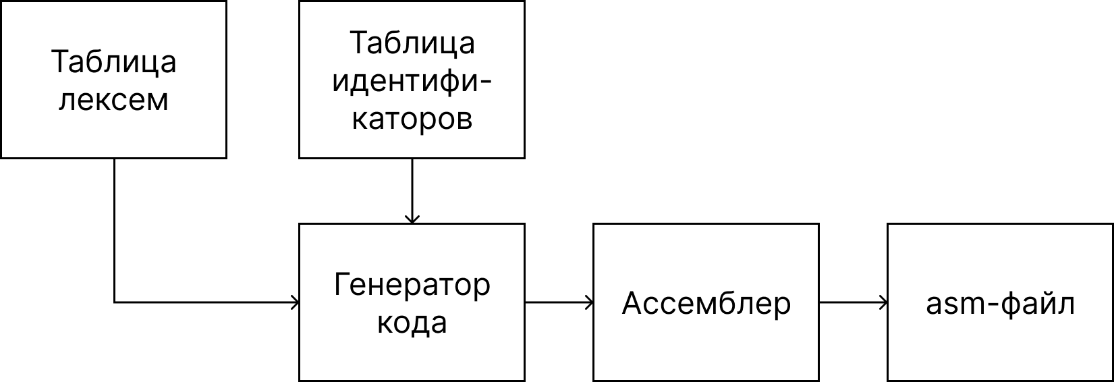


Рисунок 7.2 – Структура генератора кода

Таблица 7.4 – перечень и описание разработанных для реализации генерации кода функций.

|  |  |
| --- | --- |
| Функция | Краткое описание |
| bool CodeGeneration(Lex::LEX& tables, Parm::PARM& parm, Log::LOG& log); | Создает, генерирует и записывает ассемблерный файл |

**7.5 Параметры управляющие генерацией кода**

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

**7.6 Контрольный пример**

Результат генерации ассемблерного кода на основе контрольного примера из приложения А приведен в приложении Д. Результат работы контрольного примера приведён на рисунке 7.3.

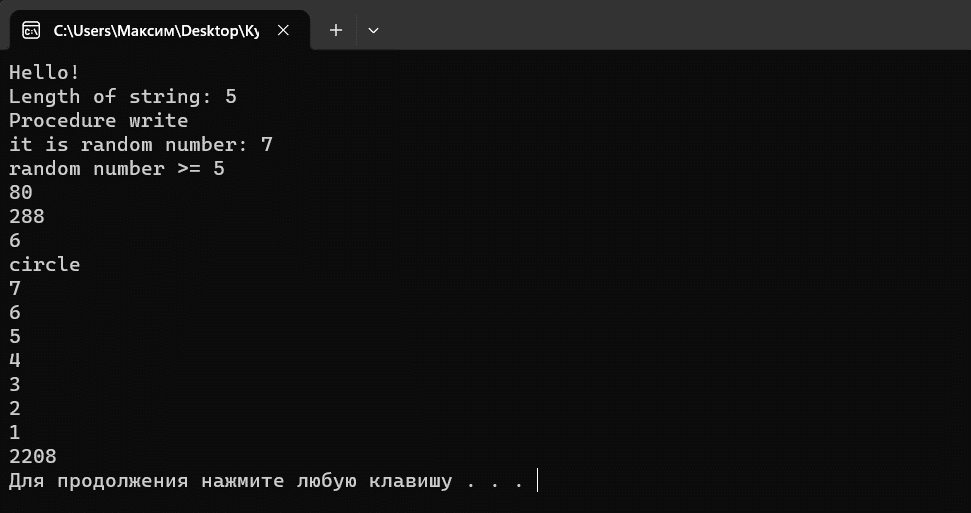


Рисунок 7.3 Результат работы программы на языке RMA-2023

**8. Тестирование транслятора**

**8.1 Общие положения**

В языке RMA-2023 не разрешается использовать запрещённые входным алфавитом символы. Результат использования запрещённого символа показан в таблице 8.1.

**8.2 Тестирование проверки на допустимость символов**

В языке RMA-2023 не разрешается использовать запрещённые входным алфавитом символы. Результат использования запрещённого символа показан в таблице 8.1.

Таблица 8.1 - Тестирование проверки на допустимость символов

|  |  |
| --- | --- |
| Исходный код | Диагностическое сообщение |
| main {й} | Ошибка N200: Лексическая ошибка: Недопустимый символ в исходном файле(-in) Строка: 1 Позиция в строке: 7 |

**8.3 Тестирование лексического анализатора**

На этапе лексического анализа в языке RMA-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  {  writel s;  } | Ошибка N304: Семантическая ошибка: В объявлении отсутствует ключевое слово type Строка: 3 Позиция в строке: 11  Ошибка N303: Семантическая ошибка: В объявлении не указан тип идентификатора Строка: 3 Позиция в строке: 11  Ошибка N300: Семантическая ошибка: Необъявленный идентификатор Строка: 3 Позиция в строке: 11 |

**8.4 Тестирование синтаксического анализатора**

На этапе синтаксического анализа в языке RMA-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 |
| int function f(){}  main{  type int a = f();} | Ошибка N 607: строка 1, позиция 18, Синтаксическая ошибка: Ошибка при конструировании return |
| string function fi([]  main[] | Ошибка N 603: строка 1, позиция 20, Синтаксическая ошибка: Ошибка в списке параметров функции |
| main{}} | Ошибка N 612: строка 1, позиция 6, Синтаксическая ошибка: Недопустимая синтаксическая конструкция |
| main{  type int a = 2++1;  } | Ошибка N 610: строка 2, позиция 13, Синтаксическая ошибка: Неверный арифметический оператор |

**8.5 Тестирование семантического анализатора**

Семантический анализ в языке RMA-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{type int a = 1/0;} | Ошибка N318: Семантическая ошибка: Деление на ноль Строка: 1 Позиция в строке: 21 |
| main{a = 5;} | Ошибка N304: Семантическая ошибка: В объявлении отсутствует ключевое слово type Строка: 1 Позиция в строке: 6  Ошибка N303: Семантическая ошибка: В объявлении не указан тип идентификатора Строка: 1 Позиция в строке: 6  Ошибка N300: Семантическая ошибка: Необъявленный идентификатор Строка: 1 Позиция в строке: 6 |
| main{}  main{} | Ошибка N302: Семантическая ошибка: Обнаружено несколько точек входа main Строка: 2 Позиция в строке: 1 |
| Main{a = 1;} | Ошибка N300: Семантическая ошибка: Необъявленный идентификатор Строка: 1 Позиция в строке: 1  Ошибка N301: Семантическая ошибка: Отсутствует точка входа main |
| main{type int t;  type string t;} | Ошибка N305: Семантическая ошибка: Попытка переопределения идентификатора Строка: 2 Позиция в строке: 13 |

**Заключение**

В ходе выполнения курсовой работы был разработан транслятор и генератор кода для языка программирования RMA-2023 со всеми необходимыми компонентами. Таким образом, были выполнены основные задачи данной курсовой работы:

* Сформулирована спецификация языка RMA-2023;
* Разработаны конечные автоматы и важные алгоритмы на их основе для эффективной работы лексического анализатора;
* Осуществлена программная реализация лексического анализатора, распознающего допустимые цепочки спроектированного языка;
* Разработана контекстно-свободная, приведённая к нормальной форме Грейбах, грамматика для описания синтаксически верных конструкций языка;
* Осуществлена программная реализация синтаксического анализатора;
* Разработан семантический анализатор, осуществляющий проверку используемых инструкций на соответствие логическим правилам;
* Разработан транслятор кода на язык ассемблера;
* Проведено тестирование всех вышеперечисленных компонентов.

Окончательная версия языка RMA-2023 включает:

* Возможность вызова функций стандартной библиотеки;
* Наличие 3 типов данных;
* Наличие 5 арифметических операторов для вычисления выражений;
* Наличие 6 логических операторов для вычисления выражений;
* Поддержка функций, процедур, операторов цикла и условия;
* Структурированная и классифицированная система для обработки ошибок пользователя;
* Поддержка операторов вывода.

Проделанная работа позволила получить необходимое представление о структурах и процессах, использующихся при построении трансляторов, а также основные различия и преимущества тех или иных средств трансляции.

**Список использованных источников**

1. Курс лекций по КПО Наркевич А.С.

2. Ахо, А. Компиляторы: принципы, технологии и инструменты / А. Ахо, Р. Сети, Дж. Ульман. – M.: Вильямс, 2003. – 768с.

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

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

5. Страуструп, Б. Принципы и практика использования C++ / Б. Страуструп – 2009 – 1238 с

**Приложение А**

|  |
| --- |
| void function FV()\процедура  {  writeline "Procedure write";  return;  }  int function randomNumber()\функция  {  type int a = random()%10;\использование стандартной библиотеки  write "it is random number: ";  writeline a;  return a;  }  main{\точка входа в программу  type symbol s = 'H';  type string str = "ello!";  write s;  writeline str;  type int l = lenght(str);  write "Length of string: ";  writeline l;  FV();  type int r = randomNumber();  if r >= 5\условная конструкция  istrue[writeline "random number >= 5";]  isfalse[writeline "random number < 5";]  type int o = 0120;  type int h = 0x120;  type int d = 0b110;  writeline o;  writeline h;  writeline d;  writeline "circle";  while r\конструкция цикла  do  [  writeline r;  r--;  ]  type int q = (o + h) \* d;  writeline q;  } |

Листинг 1 – Исходный код программы на языке RMA-2023

|  |
| --- |
| ERROR errors[ERROR\_MAX\_ENTRY] =  {  ERROR\_ENTRY(0, "Системная ошибка: Недопустимый код ошибки"),  ERROR\_ENTRY(1, "Системная ошибка: Системный сбой"),  ERROR\_ENTRY\_NODEF(2), ERROR\_ENTRY\_NODEF(3), ERROR\_ENTRY\_NODEF(4), ERROR\_ENTRY\_NODEF(5),  ERROR\_ENTRY\_NODEF(6), ERROR\_ENTRY\_NODEF(7), ERROR\_ENTRY\_NODEF(8), ERROR\_ENTRY\_NODEF(9),  ERROR\_ENTRY\_NODEF10(10), ERROR\_ENTRY\_NODEF10(20), ERROR\_ENTRY\_NODEF10(30), ERROR\_ENTRY\_NODEF10(40),  ERROR\_ENTRY\_NODEF10(50), ERROR\_ENTRY\_NODEF10(60), ERROR\_ENTRY\_NODEF10(70), ERROR\_ENTRY\_NODEF10(80), ERROR\_ENTRY\_NODEF10(90),  ERROR\_ENTRY(100, "Системная ошибка: Параметр -in должен быть задан"),  ERROR\_ENTRY(101, "Системная ошибка: Превышена длина входного параметра"),  ERROR\_ENTRY(102, "Системная ошибка: Ошибка при открытии файла с исходным кодом(-in)"),  ERROR\_ENTRY(103, "Системная ошибка: Ошибка при создании файла протокола(-log)"),  ERROR\_ENTRY\_NODEF(104), ERROR\_ENTRY\_NODEF(105), ERROR\_ENTRY\_NODEF(106), ERROR\_ENTRY\_NODEF(107), ERROR\_ENTRY\_NODEF(108), ERROR\_ENTRY\_NODEF(109),  ERROR\_ENTRY\_NODEF10(110), ERROR\_ENTRY\_NODEF10(120), ERROR\_ENTRY\_NODEF10(130), ERROR\_ENTRY\_NODEF10(140),  ERROR\_ENTRY\_NODEF10(150), ERROR\_ENTRY\_NODEF10(160), ERROR\_ENTRY\_NODEF10(170), ERROR\_ENTRY\_NODEF10(180), ERROR\_ENTRY\_NODEF10(190),  ERROR\_ENTRY(200, "Лексическая ошибка: Недопустимый символ в исходном файле(-in)"),  ERROR\_ENTRY(201, "Лексическая ошибка: Неизвестная последовательность символов"),  ERROR\_ENTRY(202, "Лексическая ошибка: Превышен размер таблицы лексем"),  ERROR\_ENTRY(203, "Лексическая ошибка: Превышен размер таблицы идентификаторов"),  ERROR\_ENTRY(204, "Лексическая ошибка : Превышен размер таблицы слов"),  ERROR\_ENTRY\_NODEF(205), ERROR\_ENTRY\_NODEF(206), ERROR\_ENTRY\_NODEF(207), ERROR\_ENTRY\_NODEF(208), ERROR\_ENTRY\_NODEF(209),  ERROR\_ENTRY\_NODEF10(210), ERROR\_ENTRY\_NODEF10(220), ERROR\_ENTRY\_NODEF10(230), ERROR\_ENTRY\_NODEF10(240),  ERROR\_ENTRY\_NODEF10(250), ERROR\_ENTRY\_NODEF10(260), ERROR\_ENTRY\_NODEF10(270), ERROR\_ENTRY\_NODEF10(280), ERROR\_ENTRY\_NODEF10(290),  ERROR\_ENTRY(300, "Семантическая ошибка: Необъявленный идентификатор"),  ERROR\_ENTRY(301, "Семантическая ошибка: Отсутствует точка входа main"),  ERROR\_ENTRY(302, "Семантическая ошибка: Обнаружено несколько точек входа main"),  ERROR\_ENTRY(303, "Семантическая ошибка: В объявлении не указан тип идентификатора"),  ERROR\_ENTRY(304, "Семантическая ошибка: В объявлении отсутствует ключевое слово type"),  ERROR\_ENTRY(305, "Семантическая ошибка: Попытка переопределения идентификатора"),  ERROR\_ENTRY(306, "Семантическая ошибка: Превышено максимальное количество параметров функции"),  ERROR\_ENTRY(307, "Семантическая ошибка: Слишком много параметров в вызове"),  ERROR\_ENTRY(308, "Семантическая ошибка: Кол-во ожидаемыех функцией и передаваемых параметров не совпадают"),  ERROR\_ENTRY(309, "Семантическая ошибка: Несовпадение типов передаваемых параметров"),  ERROR\_ENTRY(310, "Семантическая ошибка: Использование пустого строкового литерала недопустимо"),  ERROR\_ENTRY(311, "Семантическая ошибка: Обнаружен символ \'\"\'. Возможно, не закрыт строковый литерал"),  ERROR\_ENTRY(312, "Семантическая ошибка: Превышен размер литерала"),  ERROR\_ENTRY(313, "Семантическая ошибка: Недопустимый целочисленный литерал"),  ERROR\_ENTRY(314, "Семантическая ошибка: Типы данных в выражении не совпадают"),  ERROR\_ENTRY(315, "Семантическая ошибка: Тип функции и возвращаемого значения не совпадают"),  ERROR\_ENTRY(316, "Семантическая ошибка: Недопустимое строковое выражение справа от знака \'=\'"),  ERROR\_ENTRY(317, "Семантическая ошибка: Неверное условное выражение"),  ERROR\_ENTRY(318, "Семантическая ошибка: Деление на ноль"),  ERROR\_ENTRY(319, "Семантическая ошибка: Недопустимое имя переменной,использование ключевого слова"),  ERROR\_ENTRY(320, "Семантическая ошибка: Недопустим return в main"),  ERROR\_ENTRY(321, "Семантическая ошибка: Превышен максимальный размер токена"),  ERROR\_ENTRY\_NODEF(322), ERROR\_ENTRY\_NODEF(323), ERROR\_ENTRY\_NODEF(324), ERROR\_ENTRY\_NODEF(325),  ERROR\_ENTRY\_NODEF(326), ERROR\_ENTRY\_NODEF(327), ERROR\_ENTRY\_NODEF(328), ERROR\_ENTRY\_NODEF(329),  ERROR\_ENTRY\_NODEF10(330),ERROR\_ENTRY\_NODEF10(340),ERROR\_ENTRY\_NODEF10(350),  ERROR\_ENTRY\_NODEF10(360),ERROR\_ENTRY\_NODEF10(370),ERROR\_ENTRY\_NODEF10(380),ERROR\_ENTRY\_NODEF10(390),  ERROR\_ENTRY\_NODEF100(400), ERROR\_ENTRY\_NODEF100(500),  ERROR\_ENTRY(600, "Синтаксическая ошибка: Неверная структура программы"),  ERROR\_ENTRY(601, "Синтаксическая ошибка: Не найден список параметров функции"),  ERROR\_ENTRY(602, "Синтаксическая ошибка: Ошибка в теле функции"),  ERROR\_ENTRY(603, "Синтаксическая ошибка: Ошибка в списке параметров функции"),  ERROR\_ENTRY(604, "Синтаксическая ошибка: Ошибка в вызове функции/выражении"),  ERROR\_ENTRY(605, "Синтаксическая ошибка: Ошибка в списке фактических параметров функции"),  ERROR\_ENTRY(606, "Синтаксическая ошибка: Ошибка при констуировании условного выражения"),  ERROR\_ENTRY(607, "Синтаксическая ошибка: Ошибка при конструировании return"),  ERROR\_ENTRY(608, "Синтаксическая ошибка: Ошибка в условии цикла/условного выражения"),  ERROR\_ENTRY(609, "Синтаксическая ошибка: Неверный условный оператор"),  ERROR\_ENTRY(610, "Синтаксическая ошибка: Неверный арифметический оператор"),  ERROR\_ENTRY(611, "Синтаксическая ошибка: Ошибка в арифметическом выражении"),  ERROR\_ENTRY(612, "Синтаксическая ошибка: Недопустимая синтаксическая конструкция"),  ERROR\_ENTRY(613, "Синтаксическая ошибка: Недопустимая синтаксическая конструкция в теле цикла/условного выражения"),  ERROR\_ENTRY\_NODEF(614),  ERROR\_ENTRY(615, "Синтаксическая ошибка: Ожидался литерал или идентификатор"),  ERROR\_ENTRY(616, "Синтаксическая ошибка: Ошибка в теле условного выражения"),  ERROR\_ENTRY(617, "Синтаксическая ошибка: Ошибка в теле цикла"),  ERROR\_ENTRY\_NODEF10(620), ERROR\_ENTRY\_NODEF10(630), ERROR\_ENTRY\_NODEF10(640), ERROR\_ENTRY\_NODEF10(650),  ERROR\_ENTRY\_NODEF10(660), ERROR\_ENTRY\_NODEF10(670), ERROR\_ENTRY\_NODEF10(680), ERROR\_ENTRY\_NODEF10(690),  ERROR\_ENTRY\_NODEF100(700), ERROR\_ENTRY\_NODEF100(800), ERROR\_ENTRY\_NODEF100(900)  }; |

Листинг 2 – Таблица ошибок языка RMA-2023

**Приложение Б**

|  |
| --- |
| | N |СТРОКА В ТЛ | ТИП ИДЕНТИФИКАТОРА | ИМЯ | ЗНАЧЕНИЕ (ПАРАМЕТРЫ)  | 0 | 2 | function | \_FV |  | 1 | 7 | string literal | L1 |[15]Procedure write  | 2 | 14 | int function | \_randomNum |  | 3 | 20 | int variable | \_randomNua |0  | 4 | 22 | int LIB FUNC | random |  | 5 | 26 | int literal | L2 |10  | 6 | 29 | string literal | L3 |[21]it is random number:  | 7 | 42 | symbol variable | mains |  | 8 | 44 | symbol literal | L4 |H  | 9 | 48 | string variable | mainstr |[0]  | 10 | 50 | string literal | L5 |[5]ello!  | 11 | 60 | int variable | mainl |0  | 12 | 62 | int LIB FUNC | lenght |  | 13 | 68 | string literal | L6 |[18]Length of string:  | 14 | 79 | int variable | mainr |0  | 15 | 88 | int literal | L7 |5  | 16 | 92 | string literal | L8 |[18]random number >= 5  | 17 | 98 | string literal | L9 |[17]random number < 5  | 18 | 103 | int variable | maino |0  | 19 | 105 | int literal | L10 |80  | 20 | 109 | int variable | mainh |0  | 21 | 111 | int literal | L11 |288  | 22 | 115 | int variable | maind |0  | 23 | 117 | int literal | L12 |6  | 24 | 129 | string literal | L13 |[6]circle  | 25 | 139 | int UNDEFINED | :- |  | 26 | 144 | int variable | mainq |0 |

Листинг 1 – Таблица идентификаторов контрольного примера

**Приложение В**

|  |
| --- |
| Greibach greibach(NS('S'), TS('$'),  19,  Rule(NS('S'), GRB\_ERROR\_SERIES, 3,  Rule::Chain(6, TS('t'), TS('f'), TS('i'), NS('P'), NS('T'), NS('S')),  Rule::Chain(6, TS('g'), TS('f'), TS('i'), NS('P'), NS('G'), NS('S')),  Rule::Chain(4, TS('m'), TS('{'), NS('K'), TS('}'))  ),  Rule(NS('T'), GRB\_ERROR\_SERIES + 2, 2,  Rule::Chain(3, TS('{'), NS('Q'), TS('}')),  Rule::Chain(4, TS('{'), NS('K'), NS('Q'), TS('}'))  ),  Rule(NS('P'), GRB\_ERROR\_SERIES + 1, 2,  Rule::Chain(3, TS('('), NS('E'), TS(')')),  Rule::Chain(2, TS('('), TS(')'))  ),  Rule(NS('E'), GRB\_ERROR\_SERIES + 3, 2,  Rule::Chain(4, TS('t'), TS('i'), TS(','), NS('E')),  Rule::Chain(2, TS('t'), TS('i'))  ),  Rule(NS('F'), GRB\_ERROR\_SERIES + 4, 2,  Rule::Chain(3, TS('('), NS('N'), TS(')')),  Rule::Chain(2, TS('('), TS(')'))  ),  Rule(NS('N'), GRB\_ERROR\_SERIES + 5, 4,  Rule::Chain(1, TS('i')),  Rule::Chain(1, TS('l')),  Rule::Chain(3, TS('i'), TS(','), NS('N')),  Rule::Chain(3, TS('l'), TS(','), NS('N'))  ),  Rule(NS('R'), GRB\_ERROR\_SERIES + 6, 4,  Rule::Chain(2, TS('r'), NS('Y')),  Rule::Chain(2, TS('w'), NS('Y')),  Rule::Chain(4, TS('r'), NS('Y'), TS('w'), NS('Y')),  Rule::Chain(4, TS('w'), NS('Y'), TS('r'), NS('Y'))  ),  Rule(NS('Z'), GRB\_ERROR\_SERIES + 8, 5,  Rule::Chain(3, TS('i'), NS('L'), TS('i')),  Rule::Chain(3, TS('i'), NS('L'), TS('l')),  Rule::Chain(3, TS('l'), NS('L'), TS('l')),  Rule::Chain(3, TS('l'), NS('L'), TS('i')),  Rule::Chain(1, TS('i'))  ),  Rule(NS('L'), GRB\_ERROR\_SERIES + 9, 6,  Rule::Chain(1, TS('<')),  Rule::Chain(1, TS('>')),  Rule::Chain(1, TS('&')),  Rule::Chain(1, TS('!')),  Rule::Chain(1, TS('x')),  Rule::Chain(1, TS('z'))  ),  Rule(NS('A'), GRB\_ERROR\_SERIES + 10, 5,  Rule::Chain(1, TS('+')),  Rule::Chain(1, TS('-')),  Rule::Chain(1, TS('\*')),  Rule::Chain(1, TS('%')),  Rule::Chain(1, TS('/'))  ),  Rule(NS('W'), GRB\_ERROR\_SERIES + 11, 12,  Rule::Chain(1, TS('i')),  Rule::Chain(1, TS('l')),  Rule::Chain(3, TS('('), NS('W'), TS(')')),  Rule::Chain(4, TS('('), TS('-'), TS('i'), TS(')')),  Rule::Chain(6, TS('('), TS('-'), TS('i'), TS(')'), NS('A'), NS('W')),  Rule::Chain(5, TS('('), NS('W'), TS(')'), NS('A'), NS('W')),  Rule::Chain(2, TS('i'), NS('F')),  Rule::Chain(2, TS('p'), NS('F')),  Rule::Chain(3, TS('i'), NS('A'), NS('W')),  Rule::Chain(3, TS('l'), NS('A'), NS('W')),  Rule::Chain(4, TS('i'), NS('F'), NS('A'), NS('W')),  Rule::Chain(4, TS('p'), NS('F'), NS('A'), NS('W'))  ),  Rule(NS('K'), GRB\_ERROR\_SERIES + 12, 22,  Rule::Chain(7, TS('n'), TS('t'), TS('i'), TS('='), NS('V'), TS(';'), NS('K')),  Rule::Chain(7, TS('n'), TS('t'), TS('i'), TS('='), NS('W'), TS(';'), NS('K')),  Rule::Chain(5, TS('n'), TS('t'), TS('i'), TS(';'), NS('K')),  Rule::Chain(5, TS('i'), TS('='), NS('W'), TS(';'), NS('K')),  Rule::Chain(4, TS('i'), TS(':'), TS(';'), NS('K')),  Rule::Chain(4, TS('o'), NS('V'), TS(';'), NS('K')),  Rule::Chain(4, TS('^'), NS('V'), TS(';'), NS('K')),  Rule::Chain(4, TS('?'), NS('Z'), NS('R'), NS('K')),  Rule::Chain(5, TS('c'), NS('Z'), TS('d'), NS('H'), NS('K')),  Rule::Chain(6, TS('i'), TS('='), TS('U'), NS('F'), TS(';'), NS('K')),  Rule::Chain(4, TS('i'), NS('F'), TS(';'), NS('K')),  Rule::Chain(6, TS('n'), TS('t'), TS('i'), TS('='), NS('V'), TS(';')),  Rule::Chain(6, TS('n'), TS('t'), TS('i'), TS('='), NS('W'), TS(';')),  Rule::Chain(4, TS('i'), TS('='), NS('W'), TS(';')),  Rule::Chain(3, TS('i'), TS(':'), TS(';')),  Rule::Chain(4, TS('n'), TS('t'), TS('i'), TS(';')),  Rule::Chain(3, TS('o'), NS('V'), TS(';')),  Rule::Chain(3, TS('^'), NS('V'), TS(';')),  Rule::Chain(3, TS('?'), NS('Z'), NS('R')),  Rule::Chain(4, TS('c'), NS('Z'), TS('d'), NS('H')),  Rule::Chain(5, TS('i'), TS('='), TS('U'), NS('F'), TS(';')),  Rule::Chain(3, TS('i'), NS('F'), TS(';'))  ),  Rule(NS('X'), GRB\_ERROR\_SERIES + 13, 12,  Rule::Chain(5, TS('i'), TS('='), NS('W'), TS(';'), NS('X')),  Rule::Chain(4, TS('i'), TS(':'), TS(';'), NS('X')),  Rule::Chain(4, TS('o'), NS('V'), TS(';'), NS('X')),  Rule::Chain(4, TS('^'), NS('V'), TS(';'), NS('X')),  Rule::Chain(6, TS('i'), TS('='), TS('U'), NS('F'), TS(';'), NS('K')),  Rule::Chain(4, TS('i'), NS('F'), TS(';'), NS('K')),  Rule::Chain(4, TS('i'), TS('='), NS('W'), TS(';')),  Rule::Chain(3, TS('o'), NS('V'), TS(';')),  Rule::Chain(3, TS('^'), NS('V'), TS(';')),  Rule::Chain(3, TS('i'), TS(':'), TS(';')),  Rule::Chain(5, TS('i'), TS('='), TS('U'), NS('F'), TS(';')),  Rule::Chain(3, TS('i'), NS('F'), TS(';'))  ),  Rule(NS('Q'), GRB\_ERROR\_SERIES + 7, 5,  Rule::Chain(3, TS('e'), NS('V'), TS(';')),  Rule::Chain(6, TS('e'), TS('('), TS('-'), TS('i'), TS(')'), TS(';')),  Rule::Chain(2, TS('e'), TS(';')),  Rule::Chain(5, TS('e'), TS('('), TS('l'), TS(')'), TS(';')),  Rule::Chain(3, TS('e'), NS('W'), NS(';'))  ),  Rule(NS('V'), GRB\_ERROR\_SERIES + 15, 2,  Rule::Chain(1, TS('i')),  Rule::Chain(1, TS('l'))  ),  Rule(NS('Y'), GRB\_ERROR\_SERIES + 16, 3,  Rule::Chain(4, TS('['), NS('X'), NS('Q'), TS(']')),  Rule::Chain(3, TS('['), NS('X'), TS(']')),  Rule::Chain(3, TS('['), NS('Q'), TS(']'))  ),  Rule(NS('U'), GRB\_ERROR\_SERIES + 3, 2,  Rule::Chain(1, TS('i')),  Rule::Chain(1, TS('p'))  ),  Rule(NS('G'), GRB\_ERROR\_SERIES + 2, 2,  Rule::Chain(4, TS('{'), TS('e'), TS(';'), TS('}')),  Rule::Chain(5, TS('{'), NS('K'), TS('e'), TS(';'), TS('}'))  ),  Rule(NS('H'), GRB\_ERROR\_SERIES + 17, 1,  Rule::Chain(3, TS('['), NS('X'), TS(']'))  )  ); |

Листинг 1 – Грамматика языка RMA-2023

|  |
| --- |
| struct Mfst  {  enum RC\_STEP {  NS\_OK,  NS\_NORULE,  NS\_NORULECHAIN,  NS\_ERROR,  TS\_OK,  TS\_NOK,  LENTA\_END,  SURPRISE  };  struct MfstDiagnosis  {  short lenta\_position;  RC\_STEP rc\_step;  short nrule;  short nrule\_chain;  MfstDiagnosis();  MfstDiagnosis(  short plenta\_position,  RC\_STEP prt\_step,  short pnrule,  short pnrule\_chain  );  } diagnosis[MFST\_DIAGN\_NUMBER];  GRBALPHABET\* lenta;  short lenta\_position;  short nrule;  short nrulechain;  short lenta\_size;  GRB::Greibach grebach;  Lex::LEX lex;  MFSTSTACK st;  std::stack<MfstState> storestate;  Mfst();  Mfst(  Lex::LEX plex,  GRB::Greibach pgrebach  );  char\* getCSt(char\* buf);  char\* getCLenta(char\* buf, short pos, short n = 25);  char\* getDiagnosis(short n, char\* buf);  bool savestate(const Log::LOG& log);  bool reststate(const Log::LOG& log);  bool push\_chain(  GRB::Rule::Chain chain  );  RC\_STEP step(const Log::LOG& log);  bool start(const Log::LOG& log);  bool savediagnosis(  RC\_STEP pprc\_step  );  void printrules(const Log::LOG& log);  struct Deducation  {  short size;  short\* nrules;  short\* nrulechains;  Deducation() { size = 0; nrules = 0; nrulechains = 0; };  } deducation;  bool savededucation();  }; |

Листинг 2 – Структура магазинного автомата

|  |
| --- |
| struct Greibach  {  short size;  GRBALPHABET startN;  GRBALPHABET stbottomT;  Rule\* rules;  Greibach() { short size = 0; startN = 0; stbottomT = 0; rules = 0; };  Greibach(  GRBALPHABET pstartN,  GRBALPHABET pstbottomT,  short psize,  Rule r, ...  );  short getRule(  GRBALPHABET pnn,  Rule& prule  );  Rule getRule(short n);  }; |

Листинг 3 – Структура грамматики Грейбах

|  |
| --- |
| Шаг :Правило Входная лента Стек  0 :S->gfiPGS gfi(){^l;e;}tfi(){nti=p() S$  1 : SAVESTATE: 1  1 : gfi(){^l;e;}tfi(){nti=p() gfiPGS$  2 : fi(){^l;e;}tfi(){nti=p()% fiPGS$  3 : i(){^l;e;}tfi(){nti=p()%l iPGS$  4 : (){^l;e;}tfi(){nti=p()%l; PGS$  5 :P->(E) (){^l;e;}tfi(){nti=p()%l; PGS$  6 : SAVESTATE: 2  6 : (){^l;e;}tfi(){nti=p()%l; (E)GS$  7 : ){^l;e;}tfi(){nti=p()%l;o E)GS$  8 : TNS\_NORULECHAIN/NS\_NORULE  8 : RESSTATE  8 : (){^l;e;}tfi(){nti=p()%l; PGS$  9 :P->() (){^l;e;}tfi(){nti=p()%l; PGS$  10 : SAVESTATE: 2 |

Листинг 4 – Разбор исходного кода синтаксическим анализатором

|  |
| --- |
| 589 :X->i:;X i:;]} X]}$  590 : SAVESTATE: 71  590 : i:;]} i:;X]}$  591 : :;]} :;X]}$  592 : ;]} ;X]}$  593 : ]} X]}$  594 : TNS\_NORULECHAIN/NS\_NORULE  594 : RESSTATE  594 : i:;]} X]}$  595 :X->i=UF;K i:;]} X]}$  596 : SAVESTATE: 71  596 : i:;]} i=UF;K]}$  597 : :;]} =UF;K]}$  598 : TS\_NOK/NS\_NORULECHAIN  598 : RESSTATE  598 : i:;]} X]}$  599 :X->iF;K i:;]} X]}$  600 : SAVESTATE: 71  600 : i:;]} iF;K]}$  601 : :;]} F;K]}$  602 : TNS\_NORULECHAIN/NS\_NORULE  602 : RESSTATE  602 : i:;]} X]}$  603 :X->i=W; i:;]} X]}$  604 : SAVESTATE: 71  604 : i:;]} i=W;]}$  605 : :;]} =W;]}$  606 : TS\_NOK/NS\_NORULECHAIN  606 : RESSTATE  606 : i:;]} X]}$  607 :X->i:; i:;]} X]}$  608 : SAVESTATE: 71  608 : i:;]} i:;]}$  609 : :;]} :;]}$  610 : ;]} ;]}$  611 : ]} ]}$  612 : } }$  613 : $  614 : LENTA\_END  615 : ------>LENTA\_END |

Листинг 4 (продолжение) – Разбор исходного кода синтаксическим анализатором

**Приложение Г**

|  |
| --- |
| int PolishNotation(int lextable\_pos, Lex::LEX& lex)  {  stack<LT::Entry> stack;  queue<LT::Entry> queue;  LT::Entry temp; temp.idxTI = -1; temp.lexema = '#'; temp.sn = -1;  LT::Entry func; func.idxTI = -1; func.lexema = '@'; func.sn = -1;  LT::Entry tilda; tilda.idxTI = -1; tilda.lexema = '~'; tilda.sn = -1;  int countLex = 0;  int countParm = -1;  int posLex = lextable\_pos;  bool findFunc = false;  bool findComma = false;  bool flagthethis = false;  int skob = 0;  int comma = 0;  char\* buf = new char[2];  for (int i = lextable\_pos; lex.lextable.table[i].lexema != LEX\_SEPARATOR; i++, countLex++)  {  switch (lex.lextable.table[i].lexema)  {  case LEX\_STDFUNC:  case LEX\_ID:  {  if (lex.idtable.table[lex.lextable.table[i].idxTI].idtype == IT::F || lex.idtable.table[lex.lextable.table[i].idxTI].idtype == IT::S)  {  findFunc = true;  }  if (findFunc)  {  countParm++;  }  queue.push(lex.lextable.table[i]);  continue;  }  case LEX\_LITERAL:  {  queue.push(lex.lextable.table[i]);  if (findFunc)  {  countParm++;  }  continue;  }  case LEX\_LEFTHESIS:  {  if (lex.lextable.table[i + 1].lexema == LEX\_MINUS && lex.lextable.table[i + 2].lexema == LEX\_ID)  {  flagthethis = true;  continue;  }  stack.push(lex.lextable.table[i]);  continue;  }  case LEX\_RIGHTTHESIS:  {  if (findFunc)  {  func.sn = lex.lextable.table[i].sn;  lex.lextable.table[i] = func;  queue.push(lex.lextable.table[i]);  \_itoa\_s(countParm, buf, 2, 10);  stack.top().lexema = buf[0];  stack.top().idxTI = -1; stack.top().sn = lex.lextable.table[i].sn;  queue.push(stack.top());  findFunc = false;  }  else {  while (stack.top().lexema != LEX\_LEFTHESIS)  {  queue.push(stack.top());  stack.pop();  if (stack.empty())  return 0;  }  }  stack.pop();  continue;  }  case LEX\_PLUS:  case LEX\_MINUS:  case LEX\_STAR:  case LEX\_DIRSLASH:  case LEX\_PROCENT:  {  if (flagthethis)  {  tilda.sn = lex.lextable.table[i].sn;  lex.lextable.table[i] = tilda;  queue.push(lex.lextable.table[i + 1]);  queue.push(lex.lextable.table[i]);  flagthethis = false;  i += 2;  countLex += 2;  continue;  }  while (!stack.empty() && getPriority(lex.lextable.table[i].lexema) <= getPriority(stack.top().lexema))    {  queue.push(stack.top());  stack.pop();  }  stack.push(lex.lextable.table[i]);  continue;  }  case LEX\_COMMA:  {  findComma = true;  continue;  }  }  }  while (!stack.empty())  {  if (stack.top().lexema == LEX\_LEFTHESIS || stack.top().lexema == LEX\_RIGHTTHESIS)  return 0;  queue.push(stack.top());  stack.pop();  }  while (countLex != 0)  {  if (!queue.empty()) {  lex.lextable.table[posLex++] = queue.front();    queue.pop();  }  else  {  lex.lextable.table[posLex++] = temp;  }  countLex--;  }  for (int i = 0; i < posLex; i++)  {  if (lex.lextable.table[i].lexema == LEX\_LITERAL)  lex.idtable.table[lex.lextable.table[i].idxTI].idxfirstLE = i;  }  return lex.lextable.size;  } |

Листинг 1 – Программная реализация механизма преобразования в ПОЛИЗ

|  |
| --- |
| | N |СТРОКА В ТЛ | ТИП ИДЕНТИФИКАТОРА | ИМЯ | ЗНАЧЕНИЕ (ПАРАМЕТРЫ)  |0 | 2 | function | \_FV |  |1 | 7 | string literal | L1 |[15]Procedure write  |2 | 14 | int function | \_randomNum |  |3 | 20 | int variable | \_randomNua |0  |4 | 22 | int LIB FUNC | random |  |5 | 25 | int literal | L2 |10  |6 | 29 | string literal | L3 |[21]it is random number:  |7 | 42 | symbol variable | mains |  |8 | 44 | symbol literal | L4 |H  |9 | 48 | string variable | mainstr |[0]  |10 | 50 | string literal | L5 |[5]ello!  |11 | 60 | int variable | mainl |0  |12 | 62 | int LIB FUNC | lenght |  |13 | 68 | string literal | L6 |[18]Length of string:  |14 | 79 | int variable | mainr |0  |15 | 88 | int literal | L7 |5  |16 | 92 | string literal | L8 |[18]random number >= 5  |17 | 98 | string literal | L9 |[17]random number < 5  |18 | 103 | int variable | maino |0  |19 | 105 | int literal | L10 |80  |20 | 109 | int variable | mainh |0  |21 | 111 | int literal | L11 |288  |22 | 115 | int variable | maind |0  |23 | 117 | int literal | L12 |6  |24 | 129 | string literal | L13 |[6]circle  |25 | 139 | int UNDEFINED | :- | |

Листинг 2 – Таблица идентификаторов после преобразования выражений в ПОЛИЗ

|  |  |
| --- | --- |
| | N | ЛЕКСЕМА | СТРОКА | ИНДЕКС В ТИ |  | 0 | g | 1 |  | 1 | f | 1 |  | 2 | i | 1 | 0  | 3 | ( | 1 |  | 4 | ) | 1 |  | 5 | { | 2 |  | 6 | ^ | 3 |  | 7 | l | 3 | 1  | 8 | ; | 3 |  | 9 | e | 4 |  | 10 | ; | 4 |  | 11 | } | 5 |  | 12 | t | 7 |  | 13 | f | 7 |  | 14 | i | 7 | 2  | 15 | ( | 7 |  | 16 | ) | 7 |  | 17 | { | 8 |  | 18 | n | 9 |  | 19 | t | 9 |  | 20 | i | 9 | 3  | 21 | = | 9 |  | 22 | p | 9 | 4  | 23 | ( | 9 |  | 24 | ) | 9 |  | 25 | % | 9 |  | 26 | l | 9 | 5  | 27 | ; | 9 |  | 28 | o | 10 |  | 29 | l | 10 | 6  | 30 | ; | 10 |  | 31 | ^ | 11 |  | 32 | i | 11 | 3  | 33 | ; | 11 |  | 34 | e | 12 |  | 35 | i | 12 | 3  | 36 | ; | 12 |  | 37 | } | 13 |  | 38 | m | 15 |  | 39 | { | 15 |  | 40 | n | 16 |  | 41 | t | 16 |  | 42 | i | 16 | 7  | 43 | = | 16 |  | 44 | l | 16 | 8  | 45 | ; | 16 |  | 46 | n | 17 |  | 47 | t | 17 |  | 48 | i | 17 | 9  | 49 | = | 17 |  | 50 | l | 17 | 10  | 51 | ; | 17 | | 52 | o | 18 |  | 53 | i | 18 | 7  | 54 | ; | 18 |  | 55 | ^ | 19 |  | 56 | i | 19 | 9  | 57 | ; | 19 |  | 58 | n | 20 |  | 59 | t | 20 |  | 60 | i | 20 | 11  | 61 | = | 20 |  | 62 | p | 20 | 12  | 63 | ( | 20 |  | 64 | i | 20 | 9  | 65 | ) | 20 |  | 66 | ; | 20 |  | 67 | o | 21 |  | 68 | l | 21 | 13  | 69 | ; | 21 |  | 70 | ^ | 22 |  | 71 | i | 22 | 11  | 72 | ; | 22 |  | 73 | i | 23 | 0  | 74 | ( | 23 |  | 75 | ) | 23 |  | 76 | ; | 23 |  | 77 | n | 24 |  | 78 | t | 24 |  | 79 | i | 24 | 14  | 80 | = | 24 |  | 81 | i | 24 | 2  | 82 | ( | 24 |  | 83 | ) | 24 |  | 84 | ; | 24 |  | 85 | ? | 25 |  | 86 | i | 25 | 14  | 87 | x | 25 |  | 88 | l | 25 | 15  | 89 | r | 26 |  | 90 | [ | 26 |  | 91 | ^ | 26 |  | 92 | l | 26 | 16  | 93 | ; | 26 |  | 94 | ] | 26 |  | 95 | w | 27 |  | 96 | [ | 27 |  | 97 | ^ | 27 |  | 98 | l | 27 | 17  | 99 | ; | 27 |  |100 | ] | 27 |  |101 | n | 28 |  |102 | t | 28 | | |103 | i | 28 | 18  |104 | = | 28 |  |105 | l | 28 | 19  |106 | ; | 28 |  |107 | n | 29 |  |108 | t | 29 |  |109 | i | 29 | 20  |110 | = | 29 |  |111 | l | 29 | 21  |112 | ; | 29 |  |113 | n | 30 |  |114 | t | 30 |  |115 | i | 30 | 22  |116 | = | 30 |  |117 | l | 30 | 23  |118 | ; | 30 |  |119 | ^ | 31 |  |120 | i | 31 | 18  |121 | ; | 31 |  |122 | ^ | 32 |  |123 | i | 32 | 20  |124 | ; | 32 |  |125 | ^ | 33 |  |126 | i | 33 | 22  |127 | ; | 33 |  |128 | ^ | 34 |  |129 | l | 34 | 24  |130 | ; | 34 |  |131 | c | 35 |  |132 | i | 35 | 14  |133 | d | 36 |  |134 | [ | 37 |  |135 | ^ | 38 |  |136 | i | 38 | 14  |137 | ; | 38 |  |138 | i | 39 | 14  |139 | : | 39 | 25  |140 | ; | 39 |  |141 | ] | 40 |  |142 | n | 41 |  |143 | t | 41 |  |144 | i | 41 | 26  |145 | = | 41 |  |146 | ( | 41 |  |147 | i | 41 | 18  |148 | + | 41 |  |149 | i | 41 | 20  |150 | ) | 41 |  |151 | \* | 41 |  |152 | i | 41 | 22  |153 | ; | 41 |  |154 | ^ | 42 |  |155 | i | 42 | 26  |156 | ; | 42 |  |157 | } | 43 | |

Листинг 3 – Таблица лексем после преобразования выражений в ПОЛИЗ

**Приложение Д**

|  |  |
| --- | --- |
| .586  .model flat, stdcall  includelib libucrt.lib  includelib kernel32.lib  includelib ../RMA-2023\_LIB.lib  ExitProcess PROTO:DWORD  .stack 4096  outnum PROTO : DWORD  outstr PROTO : DWORD  outstrline PROTO : DWORD  outnumline PROTO : DWORD  system\_pause PROTO  random PROTO  lenght PROTO : DWORD  .const  null\_division BYTE 'ERROR: DIVISION BY ZERO', 0  overflow BYTE 'ERROR: VARIABLE OVERFLOW', 0  L1 BYTE 'Procedure write', 0  L2 SWORD 10  L3 BYTE 'it is random number: ', 0  L4 BYTE 'H', 0  L5 BYTE 'ello!', 0  L6 BYTE 'Length of string: ', 0  L7 SWORD 5  L8 BYTE 'random number >= 5', 0  L9 BYTE 'random number < 5', 0  L10 SWORD 80  L11 SWORD 288  L12 SWORD 6  L13 BYTE 'circle', 0  .data  \_randomNua SWORD 0  mainmov SWORD 0  mains DWORD ?  mainstr DWORD ?  mainl SWORD 0  mainr SWORD 0  maino SWORD 0  mainh SWORD 0  maind SWORD 0  mainq SWORD 0  .code  \_FV PROC  push offset L1  call outstrline  ret  SOMETHINGWRONG:  push offset null\_division  call outstrline  call system\_pause  push -1  call ExitProcess  EXIT\_OVERFLOW:  push offset overflow  call outstrline  call system\_pause  push -2  call ExitProcess  \_FV ENDP  \_randomNum PROC  push random  pop dx  call random  push ax  push L2  pop bx  pop ax  cmp bx,0  je SOMETHINGWRONG  cdq  idiv bx  push dx  pop \_randomNua  push offset L3  call outstr  push \_randomNua  call outnumline  mov ax, \_randomNua  ret  SOMETHINGWRONG:  push offset null\_division  call outstrline  call system\_pause  push -1  call ExitProcess  EXIT\_OVERFLOW:  push offset overflow  call outstrline  call system\_pause  push -2  call ExitProcess  \_randomNum ENDP  main PROC  push offset L4  pop mains  push offset L5  pop mainstr  push mains  call outstr  push mainstr  call outstrline  push lenght  push mainstr  pop dx  pop dx  push mainstr  call lenght  push ax  pop mainl  push offset L6  call outstr  push mainl  call outnumline | call \_FV  push \_randomNum  pop dx  call \_randomNum  push ax  pop mainr  mov dx, mainr  cmp dx, L7  jge right1  jle wrong1  jmp next1  right1:  push offset L8  call outstrline  jmp next1  wrong1:  push offset L9  call outstrline  next1: push L10  pop maino  push L11  pop mainh  push L12  pop maind  push maino  call outnumline  push mainh  call outnumline  push maind  call outnumline  push offset L13  call outstrline  mov dx, mainr  cmp dx, 0  jnz cycle1  jmp continue1  cycle1:  push mainr  call outnumline  mov ax,mainr  mov bx,1  sub ax, bx  jo EXIT\_OVERFLOW  mov mainr , ax  jnz cycle1  continue1: push maino  push mainh  pop ax  pop bx  add ax, bx  jo EXIT\_OVERFLOW  push ax  push maind  pop ax  pop bx  imul bx  jo EXIT\_OVERFLOW  push ax  pop mainq  push mainq  call outnumline  call system\_pause  push 0  call ExitProcess  SOMETHINGWRONG:  push offset null\_division  call outstrline  call system\_pause  push -1  call ExitProcess  EXIT\_OVERFLOW:  push offset overflow  call outstrline  call system\_pause  push -2  call ExitProcess  main ENDP  end main |

Листинг 1 – Результат генерации кода контрольного примера в Ассемблере