МИНИСТЕРСТВО ОБРАЗОВАНИЯ РЕСПУБЛИКИ БЕЛАРУСЬ

Учреждение образования «БЕЛОРУССКИЙ

ГОСУДАРСТВЕННЫЙ ТЕХНОЛОГИЧЕСКИЙ УНИВЕРСИТЕТ»

Факультет информационных технологий

Кафедра программной инженерии

Специальность 6-05-0612-01 Программная инженерия

**ПОЯСНИТЕЛЬНАЯ ЗАПИСКА**

**К КУРСОВОМУ ПРОЕКТУ НА ТЕМУ:**

«Разработка компилятора RMV-2024»

Выполнил студент Рублевская Маргарита Владимировна

(Ф.И.О.)

Руководитель проекта преп.-ст. Некрасова А.П.

Заведующий кафедрой к.т.н., доц. Смелов В.В.

Консультанты преп.-ст. Некрасова А.П.

Нормоконтролер преп.-ст. Некрасова А.П.

Курсовой проект защищен с оценкой

Минск 2024

Содержание

[Введение 5](#_Toc185190157)

[1. Спецификация языка программирования 6](#_Toc185190158)

[1.1 Характеристика языка программирования 6](#_Toc185190159)

[1.2 Определение алфавита языка программирования 6](#_Toc185190160)

[1.3 Применяемые сепараторы 6](#_Toc185190161)

[1.4 Применяемые кодировки 6](#_Toc185190162)

[1.5 Типы данных 6](#_Toc185190163)

[1.6 Преобразование типов данных 7](#_Toc185190164)

[1.7 Идентификаторы 8](#_Toc185190165)

[1.8 Литералы 8](#_Toc185190166)

[1.9 Объявления данных 9](#_Toc185190167)

[1.10 Инициализация данных 9](#_Toc185190168)

[1.11 Инструкции языка 9](#_Toc185190169)

[1.12 Операции языка 10](#_Toc185190170)

[1.13 Выражения и их вычисления 10](#_Toc185190171)

[1.14 Конструкции языка 11](#_Toc185190172)

[1.15 Область видимости идентификаторов 11](#_Toc185190173)

[1.16 Семантические проверки 11](#_Toc185190174)

[1.17 Распределение оперативной памяти на этапе выполнения 12](#_Toc185190175)

[1.18 Стандартная библиотека и её состав 12](#_Toc185190176)

[1.19 Ввод и вывод данных 13](#_Toc185190177)

[1.20 Точка входа 13](#_Toc185190178)

[1.21 Препроцессор 13](#_Toc185190179)

[1.22 Соглашения о вызовах 13](#_Toc185190180)

[1.23 Объектный код 13](#_Toc185190181)

[1.24 Классификация сообщений транслятора 13](#_Toc185190182)

[1.25 Контрольный пример 14](#_Toc185190183)

[2. Структура транслятора 15](#_Toc185190184)

[2.1 Компоненты транслятора, их назначение и принципы взаимодействия 15](#_Toc185190185)

[2.2 Перечень входных параметров транслятора 16](#_Toc185190186)

[2.3 Протоколы, формируемые транслятором 16](#_Toc185190187)

[3. Разработка лексического анализатора 17](#_Toc185190188)

[3.1 Структура лексического анализатора 17](#_Toc185190189)

[3.2 Контроль входных символов 17](#_Toc185190190)

[3.3 Удаление избыточных символов 18](#_Toc185190191)

[3.4 Перечень ключевых 18](#_Toc185190192)

[3.5 Основные структуры данных 20](#_Toc185190193)

[3.6 Структура и перечень сообщений лексического анализатора 21](#_Toc185190194)

[3.7 Принцип обработки ошибок 21](#_Toc185190195)

[3.8 Параметры лексического анализатора и режимы его работы 21](#_Toc185190196)

[3.9 Алгоритм лексического анализа 21](#_Toc185190197)

[3.10 Контрольный пример 21](#_Toc185190198)

[4. Разработка синтаксического анализатора 22](#_Toc185190199)

[4.1 Структура синтаксического анализатора 22](#_Toc185190200)

[4.2 Контекстно свободная грамматика, описывающая синтаксис языка 22](#_Toc185190201)

[4.3 Построение конечного магазинного автомата 25](#_Toc185190202)

[4.4 Основные структуры данных 26](#_Toc185190203)

[4.5 Описание алгоритма синтаксического разбора 26](#_Toc185190204)

[4.6 Структура и перечень сообщений синтаксического анализатора 26](#_Toc185190205)

[4.7 Параметры синтаксического анализатора и режимы его работы 26](#_Toc185190206)

[4.8 Принцип обработки ошибок 27](#_Toc185190207)

[4.9 Контрольный пример 27](#_Toc185190208)

[5. Разработка семантического анализатора 28](#_Toc185190209)

[5.1 Структура семантического анализатора 28](#_Toc185190210)

[5.2 Функции семантического анализатора 28](#_Toc185190211)

[5.3 Структура и перечень сообщений семантического анализатора 28](#_Toc185190212)

[5.4 Принцип обработки ошибок 29](#_Toc185190213)

[5.5 Контрольный пример 29](#_Toc185190214)

[6. Вычисление выражений 31](#_Toc185190215)

[6.1 Выражения, допускаемые языком 31](#_Toc185190216)

[6.2 Польская запись и принцип ее построения 31](#_Toc185190217)

[6.3 Программная реализация обработки выражений 32](#_Toc185190218)

[6.4 Контрольный пример 33](#_Toc185190219)

[7. Генерация кода 34](#_Toc185190220)

[7.1 Структура генератора кода 34](#_Toc185190221)

[7.2 Представление типов данных в оперативной памяти 34](#_Toc185190222)

[7.3 Статическая библиотека 34](#_Toc185190223)

[7.4 Особенности алгоритма генерации кода 35](#_Toc185190224)

[7.5 Входные параметры, управляющие генерацией кода 35](#_Toc185190225)

[7.6 Контрольный пример 35](#_Toc185190226)

[8. Тестирование транслятора 37](#_Toc185190227)

[8.1 Общие положения 37](#_Toc185190228)

[8.2 Результаты тестирования 37](#_Toc185190229)

[Заключение 41](#_Toc185190230)

[Список использованных литературных источников 42](#_Toc185190231)

[Приложение А 43](#_Toc185190232)

[Приложение Б 45](#_Toc185190233)

[Приложение В 47](#_Toc185190234)

[Приложение Г 51](#_Toc185190235)

[Приложение Д 53](#_Toc185190236)

[Приложение Е 55](#_Toc185190237)

[Приложение Ж 56](#_Toc185190238)

# **Введение**

Данный курсовой проект представляет из себя создание собственного языка программирования RMV-2024. Этот язык программирования предназначен для выполнения простейших операций и арифметических действий над числами.

Для создания собственного языка программирования требуется разработать собственный компилятор. Компилятор – это программа, задачей которого является перевод программы, написанной на одном из языков программирования в программу на язык ассемблера. В задачи компилятора входит: лексический анализ, семантический анализ и синтаксический анализ.

Исходя из раннее определённой 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 ассемблер;
* тестирование транслятора.

Пояснительная записка описывает правила и требования в использовании, принцип работы, реализацию разработанного языка программирования и компилятора.

1. Спецификация языка программирования

1. Характеристика языка программирования

Язык программирования RMV-2024 – высокоуровневый, процедурный, компилируемый, не объектно-ориентированный. Имеет нестрогую статическую типизацию.

1. Определение алфавита языка программирования

**Язык программирования** RMV-2024 **использует стандартную таблицу символов Windows-1251.** Исходный код RMV-2024 может включать в себя символы латинского алфавита, цифры десятичной системы счисления от 0 до 9 и некоторые специальные символы для операций и строковых литералов

1. Применяемые сепараторы

Сепараторы необходимы для разделения операций языка. Сепараторы, используемые в языке программирования RMV-2024, приведены в таблице 1.1.

Таблица 1.1 – Применяемые сепараторы

|  |  |
| --- | --- |
| Сепаратор | Назначение сепаратора |
| ; | Разделитель инструкций |
| {…} | Программный блок |
| (…) | Параметры, приоритетность операций |
| ‘пробел’ | Служит для разделения программных конструкций. Допускается везде, кроме идентификаторов и ключевых слов |
| , | Разделитель параметров в функции |
| > < >= <=  == ^= | Логические операции (операции сравнения: больше, меньше, больше или равно, меньше или равно, равно, не равно,), используемые в условии цикла/условной конструкции, а также в выражениях |
| + - \* / % | Арифметические операции (сложение, вычитание, умножение, деление, остаток от деления). Используются в выражениях |
| = | Оператор присваивания |

1. Применяемые кодировки

**Для написания исходного кода программы на языке RMV-2024 используются латинские символы и символы разделители кодировки Windows-1251.** Кириллические символы допустимы только в строковых литералах и комментариях.

1. Типы данных

В языке RMV-2024 есть три фундаментальных типа данных: целочисленный, логический и строковый. Описание типов данных, предусмотренных в данном языке представлено в таблице 1.2.

Таблица 1.2 – Типы данных языка RMV-2024

|  |  |
| --- | --- |
| Тип данных | Описание |
| Целочисленный тип данных (number) | Фундаментальный тип данных, представляющий собой целое число, занимающий 2 байта памяти. Автоматическая инициализация происходит с присвоением значения 0. Возможные значения: от -32,768 до 32,767 включительно  Возможные операции:  + (бинарный) – оператор сложения;  - (бинарный) – оператор вычитания;  \* (бинарный) – оператор умножения;  / (бинарный) – оператор целочисленного деления;  % (бинарный) – оператор остатка от деления;  > (бинарный) – оператор «больше»;  < (бинарный) – оператор «меньше»;  >= (бинарный) – оператор «больше или равно»;  <= (бинарный) – оператор «меньше или равно»;  == (бинарный) – оператор «равно»;  ^= (бинарный) – оператор «не равно».  Может применяться в качестве аргумента в функциях, условия для оператора цикла и условного оператора. |
| Логический тип данных (bool) | Фундаментальный тип данных, предусмотренный для объявления целых чисел, занимающий 1 байт памяти.  Значения: true (истина), false (ложь).  Инициализация по умолчанию: false.  Поддерживаемые операции:  **+** (бинарный) – оператор сложения;  - (бинарный) – оператор вычитания;  **\*** (бинарный) – оператор умножения;  **/** (бинарный) – оператор целочисленного деления;  % (бинарный) – оператор остатка от деления;  **=** (бинарный) – оператор присваивания. |
| Строковый тип данных (line) | Фундаментальный тип данных, предусмотренный для работы с символами, каждый из которых занимает 1 байт. Максимальное количество символов – 256.  Инициализация по умолчанию: строка нулевой длины “”. Поддерживаемые операции:  **=** (бинарный) – оператор присваивания. |

1. Преобразование типов данных

В языке программирования RMV-2024 предусмотрены неявные преобразования между логическими (bool) и целочисленными (number) типами. Значение логического типа true преобразуется к 1, значение false – к 0. К логическому значению true преобразуются все значения типа number кроме 0, он преобразуется к false.

Логический тип преобразуется к целочисленному в следующих случаях:

* при присваивании результата логического выражения переменной типа number;
* при присваивании значения логического идентификатора или литерала переменной типа number;
* при вычислении арифметических выражений;
* при передаче логического типа в качестве целочисленного аргумента функции.

Целочисленный тип преобразуется к логическому в следующих случаях:

* при присваивании результата арифметического выражения переменной типа bool;
* при присваивании значения целочисленного идентификатора или литерала переменной типа bool;
* при передаче целочисленного типа в качестве логического аргумента функции;
* при передаче целочисленного идентификатора или литерала в качестве условия для оператора цикла или условного оператора.

Преобразование строкового типа line в какой-либо другой фундаментальный тип не производится.

1. Идентификаторы

Идентификаторы в языке RMV-2024 используются для именования функций, процедур, параметров функций и переменных. Идентификаторы, объявленные внутри блока функции или процедуры получают префикс, идентичный имени функции/процедуры для разрешения области видимости. Идентификаторы не должны совпадать с ключевыми словами языка, а также с именами команд ассемблера. Максимальная длина идентификатора не должна превышать 10 символов.

* Примеры правильных идентификаторов: numA, count, factorial, getName, result, i, isNegative.

1. Литералы

В языке RMV-2024 существует 3 типа литералов: логического, строкового и целочисленного типов. Краткое описание литералов представлено в таблице 1.3.

Таблица 1.3 – Описание литералов

|  |  |
| --- | --- |
| Тип литерала | Описание литерала |
| Логические | Распознаются при помощи ключевых слов “true” и “false”, соответственно значения от 0 (false) до 1 (true). |
| Строковые | Состоит из символов, заключенных в (одинарные кавычки), инициализируются пустой строкой, строковые переменные. |
| Целочисленные | Последовательность цифр 0…9 с предшествующим знаком минус или без него.  Диапазоны значений: от -32768 до 32767. |

1. Объявления данных

В языке RMV-2024 при объявлении переменной используется ключевое слово def, за которым следует указание типа данных и имени идентификатора. Допускается инициализировать данные при объявлении. Для объявления функции используется ключевое слово function после которого также следует указание типа и имени идентификатора. При объявлении процедуры используется ключевое слово procedureпосле которого сразу следует имя процедуры без указания типа.

1. Инициализация данных

Существует несколько способов инициализации данных в программировании. Первый способ – это инициализация по умолчанию при объявлении переменной. В этом случае переменные типа bool инициализируются значением false, переменные типа number – значением 0, а переменные типа line – пустой строкой.

Второй способ инициализации предполагает присваивание значения при объявлении переменной. Здесь значение может быть представлено в виде литерала, идентификатора или результата вызова функции.

Третий способ инициализации данных позволяет присваивать значения уже существующим переменным. Как и в предыдущем случае, значение может быть литералом, идентификатором или результатом вызова функции.

1. Инструкции языка

Инструкции языка RMV-2024 представлена в таблице 1.4.

Таблица 1.4 – Инструкции языка

|  |  |  |
| --- | --- | --- |
| Инструкция | Форма записи | Пример |
| Объявление переменной | def <тип данных> <идентификатор>; | def number num; |
| Объявление переменной с явной инициализацией | def <тип данных> <идентификатор> = <значение>|<выражение>;  <значение>::= <литерал>| <идентификатор>|<вызов функции> | def bool b = true;  def number num = 3 \* 9; |
| Присваивание | <идентификатор> = <значение>|  <выражение>;  <значение>::= <литерал>| <идентификатор>|<вызов функции> | num = 23;  num = result;  num = factorial[9]; |
| Вызов внешней функции или процедуры | <идентификатор> [ <идентификатор>|  <литерал>, …]; | factorial[a];  factorial[9];  max[6, 11]; |
| Возврат из  подпрограммы | give <идентификатор> / <литерал>; | give result;  give 6; |
| Вывод данных | out [<идентификатор> / <литерал>]; | out[“Hello”]; |
| Вывод данных с новой строки | outln [<идентификатор> / <литерал>]; | outln[result] |

Таблица 1.4 (продолжение)

|  |  |  |
| --- | --- | --- |
| Условный оператор | when [<условие>]  { … }  otherwise  { … } | when [i > 0]  {  outln[“Positive”];  }  otherwise  {  outln[“Negative”];  } |

1. Операции языка

Операции сравнения — это операторы, которые используются для сравнения двух значений. Операции сравнения часто используются в условных выражениях и циклах., которые можно использовать в языке RMV-2024, представлены в таблице 1.5.

Таблица 1.5 – Операции языка

|  |  |  |
| --- | --- | --- |
| Тип операций | Операции | Приоритет |
| Логические операции | > – больше  < – меньше  == – равно  ^= – не равно  >= – больше или равно  <= – меньше или равно | -1  -1  -1  -1  -1  -1 |
| Арифметические операции | + – сложение  - – вычитание  \* – умножение  / – деление  % –остаток от деления | 2  2  3  3  3 |
| Операция запятая | , | 1 |

1. Выражения и их вычисления

Выражения в языке RMV-2024 составляются согласно следующим правилам:

– Допускается использование оператора смены приоритета только в арифметических выражениях;

– Использование логических и арифметических операторов в одном выражении не допустимо;

– Использование двух и более идущих подряд операторов (за исключением оператора смены приоритета) запрещено;

– Использование двух и более логических операторов в одном выражении запрещено;

– Вызов функции могут содержать только арифметические выражения или выражения присваивания;

– Вычисление сложных выражений (как минимум с одним оператором) внутри оператора возврата, в аргументах функции или процедуры, внутри условия цикла или условного оператора (за исключением логических операций) не производится.

1. Конструкции языка

Основные программные конструкции языка программирования RMV-2024 представлены в таблице 1.6.

Таблица 1.6 – Основные конструкции языка

|  |  |
| --- | --- |
| Конструкция | Реализация |
| Главная функция  (точка входа) | program  { … } |
| Процедура | procedure <идентификатор> [<тип данных> <идентификатор>, …]  {  …  } |
| Функция | function <тип данных> <идентификатор> [<тип данных> <идентификатор>, …]  {  …  give <выражение>  } |

1. Область видимости идентификаторов

В языке RMV-2024 переменные обязаны находится внутри программного блока функций или процедур. Переменные, объявленные в одной функции, недоступны в другой. Внутри разных областей видимости разрешено объявление переменных с одинаковыми именами.

Все переменные и параметры внутри области видимости получают префикс, который отображается в таблице идентификаторов. Объявление глобальных переменных не предусмотрено. Объявление пользовательских областей видимости также не предусмотрено.

1. Семантические проверки

В языке программирования RMV-2024 выполняются следующие семантические проверки:

* Наличие функции program – точки входа в программу;
* Единственность точки входа;
* Переопределение идентификаторов;
* Использование идентификаторов без их объявления;
* Проверка соответствия типа функции и возвращаемого параметра;
* Правильность передаваемых в функцию параметров: количество, типы;
* Правильность строковых выражений;
* Превышение размера строковых и целочисленных литералов;
* Деление на ноль в арифметических операциях;
* Проверка на вызов функции в логических выражениях;
* Корректность использования операторов в выражениях.

1. Распределение оперативной памяти на этапе выполнения

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

1. Стандартная библиотека и её состав

Функции стандартной библиотеки языка RMV-2024 реализованы на языке C++. Содержимое библиотеки и описание функций представлено в таблице 1.7.

Таблица 1.7 – Стандартная библиотека языка RMV-2024

|  |  |  |  |
| --- | --- | --- | --- |
| Имя функции | Возвращаемое значение | Принимаемые параметры | Описание |
| concat | line | line a, line b | Функция производит конкатенацию строк a и b, возвращает строку. |
| linelen | number | line a | Функция вычисляет длину строки a. |
| random | number | number min, number max | Функция возвращает случайно сгенерированное число в диапазоне [min, max]. |
| sqrt | number | number a | Функция возвращает результат вычисление квадратного корня из числа a. |
| OutLine | - | line a | Функция выводит в консоль строку a. |
| OutNumber | - | number a | Функция выводит в консоль число a. |
| OutBool | - | bool a | Функция выводит в консоль булево значение a. |

Таблица 1.7 (продолжение)

|  |  |  |  |
| --- | --- | --- | --- |
| OutLineLn | - | line a | Функция выводит в консоль строку a с переносом на новую строку. |
| OutNumberLn | - | number a | Функция выводит в консоль число a с переносом на новую строку. |
| OutBoolLn | - | bool a | Функция выводит в консоль булево значение a с переносом на новую строку. |

1. Ввод и вывод данных

Вывод данных осуществляется с помощью функцийout иoutln. Допускается использование функций с литералами и идентификаторами.

В зависимости от типа параметра определяется функция: OutLine, OutBool, OutNumber, OutLineLn, OutBoolLn, OutNumberLn которые входят в состав стандартной библиотеки и описаны в таблице 1.7.

Функции ввода данных в языке RMV-2024 не предусмотрены.

1. Точка входа

В языке RMV-2024 каждая программа должна содержать главную функцию, точку входа, с которой начнется последовательное выполнение программы. Точкой входа является функция с именем program.

1. Препроцессор

Препроцессор в языке программирования RMV-2024 не предусмотрен.

1. Соглашения о вызовах

Все параметры заноситься в стек вручную как для функций, так и для выполнения инструкций. В функцию параметры передаются, начиная с вершины стека, то есть верхняя вершина стека будет первым параметром и так далее.

1. Объектный код

Исходный код на языке RMV-2024 транслируется в язык ассемблера, а затем в объектный код.

1. Классификация сообщений транслятора

В случае возникновения ошибки в коде программы на языке RMV-2024 и выявления её транслятором в текущий файл протокола выводится сообщение. Их классификация сообщений приведена в таблице 1.8.

Таблица 1.8 – Классификация сообщений транслятора

|  |  |
| --- | --- |
| Интервал | Описание ошибок |
| 0-110 | Системные ошибки, ошибки параметров |
| 200-299 | Ошибки лексического анализа |
| 300-399 | Ошибки семантического анализа |
| 600-699 | Ошибки синтаксического анализа |
| 400 – 499,  700 – 999 | Зарезервированные коды ошибок |

1. Контрольный пример

Контрольный пример демонстрирует главные особенности языка RMV-2024: функции, процедуры, фундаментальные типы, основные инструкции и операции, использование функций статической библиотеки. Исходный код контрольного примера представлен в приложении А.

2. Структура транслятора

1. Компоненты транслятора, их назначение и принципы взаимодействия

Транслятор преобразует программу, написанную на языке RMV-2024 в программу на языке ассемблера. Компонентами транслятора являются лексический, синтаксический и семантический анализаторы, а также генератор кода на язык ассемблера. Принцип их взаимодействия представлен на рисунке 2.1.

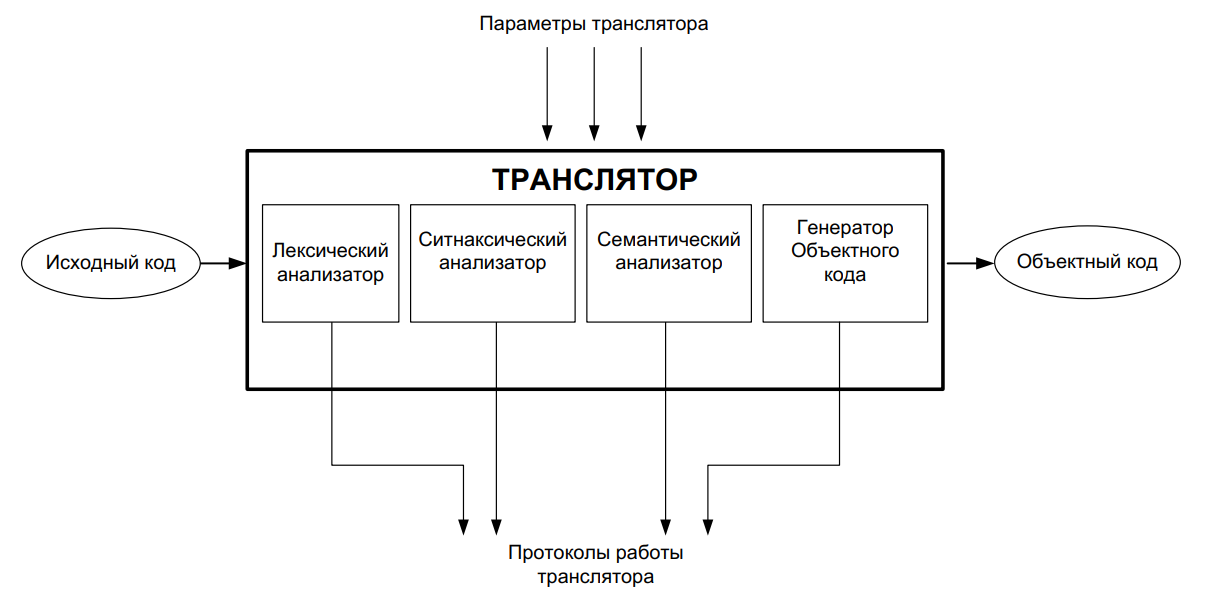


Рисунок 2.1 – Структура транслятора

Лексический анализ – первая фаза трансляции. Назначением лексического анализатора является нахождение ошибок лексики языка, удаление всех лишних пробелов, распознание лексем, формирование таблицы лексем и таблицы идентификаторов. При ошибках распознавания текста выдавать сообщение об ошибке.

Синтаксический анализ – это основная часть транслятора, предназначенная для распознавания синтаксических конструкций и формирования промежуточного кода. Входным параметром для синтаксического анализа является таблица лексем. Синтаксический анализатор распознаёт синтаксические конструкции, выявляет синтаксические ошибки при их наличии и формирует дерево разбора.

Семантический анализ в свою очередь является проверкой исходной программы на семантическую согласованность с определением языка, т.е. проверяет правильность текста исходной программы с точки зрения семантики. На вход семантического анализатора поступаются таблицы лексем и идентификаторов. Анализатор отслеживает ошибки, которые невозможно отследить при помощи регулярной и контекстно-свободной грамматики.

Генератор кода – этап транслятора, выполняющий генерацию ассемблерного кода на основе полученных данных на предыдущих этапах трансляции. Генератор кода принимает на вход таблицы идентификаторов и лексем и транслирует код на языке RMV-2024, прошедший все предыдущие этапы, в код на языке Ассемблера.

1. Перечень входных параметров транслятора

В таблице 2.1 представлены входные параметры, которые могут использоваться для управления работой транслятора.

Таблица 2.1 – Входные параметры транслятора

|  |  |  |
| --- | --- | --- |
| Входной параметр | Описание | Значение по умолчанию |
| -in:<имя\_файла> | Входной файл с расширением .txt, в котором содержится исходный код на RMV-2024 | Не предусмотрено |
| -log:<имя\_файла> | Файл для записи полного протокола работы транслятора | <имя\_файла>.log |
| -out:<имя\_файла> | Файл для записи результата работы транслятора | <имя\_файла>.out.asm |
| -poliz | Ключ для вывода на консоль промежуточного представления кода после преобразования в польскую инверсную запись | По умолчанию отсутствует |
| -lt | Ключ для вывода таблицы лексем на консоль | По умолчанию отсутствует |
| -id | Ключ для вывода таблицы идентификаторов на консоль | По умолчанию отсутствует |

1. Протоколы, формируемые транслятором

В ходе трансляции формируются протоколы работы лексического, синтаксического и семантического анализаторов. Перечень протоколов, формируемых транслятором языка программирования их описание представлены в таблице 2.2

Таблица 2.2 – Протоколы, формируемые транслятором языка RMV-2024

|  |  |
| --- | --- |
| Формируемый протокол | Описание выходного протокола |
| Файл журнала, заданный параметром -log: | Файл с протоколом работы транслятора языка программирования RMV-2024 содержит информацию о времени выполнения приложения; входных параметрах в приложение таблицу лексем, таблицу идентификаторов, промежуточное представление кода; трассировку синтаксического анализа; дерево разбора, время выполнения разбора; промежуточное представление кода после приведения его к польской нотации. |
| Выходной файл, заданный параметром -out: | Результат работы программы – файл, содержащий исходный код на языке ассемблера. |

3. Разработка лексического анализатора

1. Структура лексического анализатора

**Лексический анализатор – это часть компилятора, выполняющая лексический анализ исходного кода.** На вход лексического анализатора поступает текст исходной программы. Итогом работы лексического анализатора является список всех найденных в тексте исходной программы лексем. Этот список лексем предстаёт в виде таблицы, называемой таблицей лексем. Структура лексического анализатора представлена на рисунке 3.1.

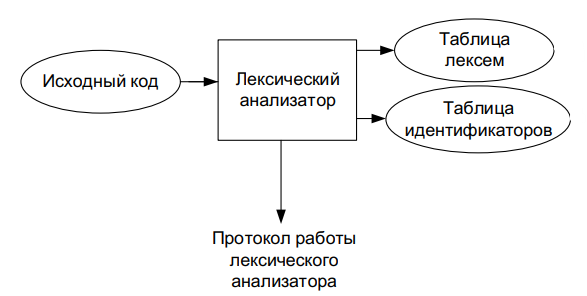
****

Рисунок 3.1 – Структура лексического анализатора

В последующем таблицы используются на других этапа анализа языка.

1. 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.

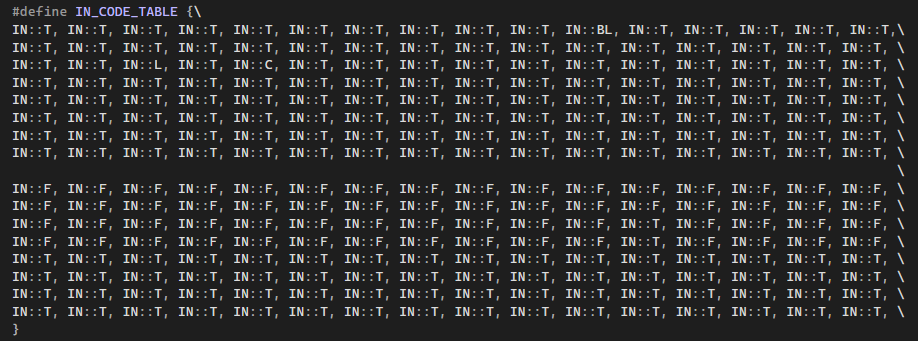


Рисунок 3.2 – Таблица контроля входных символов

Категории входных символов представлены в таблице 3.1.

Таблица 3.1 – Соответствие символов и их значений в таблице

|  |  |
| --- | --- |
| Символы | Значение в таблице входных символов |
| T | Разрешенный |
| F | Запрещенный |
| BL | Перевод строки |
| C | Признак начала комментария |
| L | Строковый литерал |

1. Удаление избыточных символов

Избыточными символами являются символы табуляции и пробелы. Они удаляются на этапе разбиения исходного кода на лексемы.

Описание алгоритма удаления избыточных символов:

* Посимвольное считывание исходного кода, занесенного в структуру In.
* Встреча пробела или знака табуляции вне пределов строкового литерала или комментария является встречей символа-сепаратора.
* В отличие от других символов-сепараторов табуляции и пробелы игнорируются, т.е. не записываются в таблицу лексем

1. Перечень ключевых

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

Таблица 3.2 – Соответствие ключевых слов, операций и сепараторов с лексемами

|  |  |  |
| --- | --- | --- |
| Тип цепочки | Цепочка | Лексема |
| Ключевые слова | def | d |
| number, line, bool | t |
| program | p |
| function | f |
| procedure | s |
| give | r |
| out | o |
| cycle | u |
| when | w |
| otherwise | ! |
| Иное | Идентификатор | i |
| Литерал | l |
| Функции стандартной библиотеки | concat | + |
| linelen | % |
| sqrt | q |
| outln | b |

Таблица 3.2 (продолжение)

|  |  |  |
| --- | --- | --- |
|  | random | z |
| Сепараторы | ; | ; |
| , | , |
| { | { |
| } | } |
| ( | ( |
| ) | ) |
| [ | [ |
| ] | ] |
| Операторы | Арифметические (+, -, \*, /, %) | v |
| Логические (== != > < >= <=) | g |
| Присваивание (=) | = |

Каждому выражению соответствует детерминированный конечный автомат, по которому происходит разбор данного выражения. 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 изображена на рисунке 3.3.

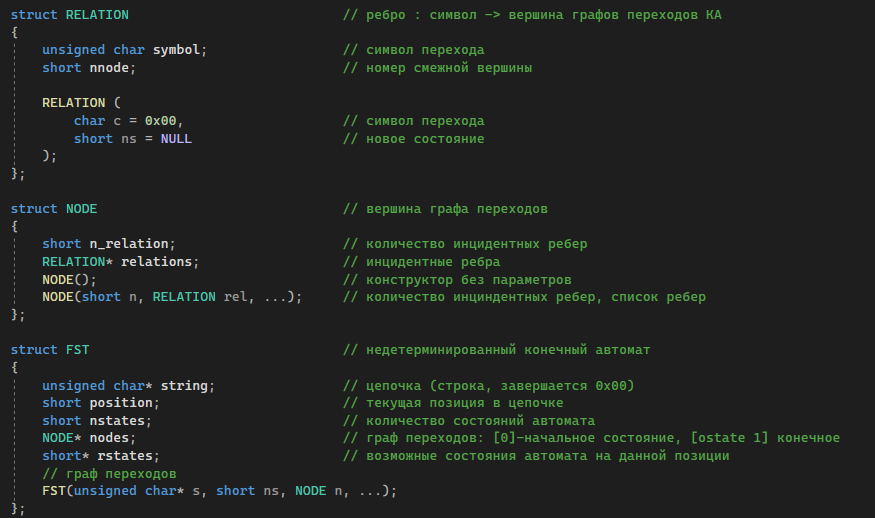


Рисунок 3.3 – Структура конечного автомата

Пример графа перехода конечного автомата изображен на рисунке 3.4.

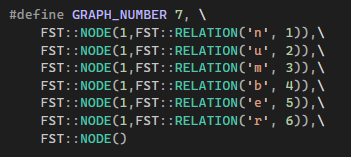


Рисунок 3.4 – Граф конечного автомата для ключевого слова number

1. Основные структуры данных

Основными структурами данных лексического анализатора являются таблица лексем и таблица идентификаторов. Код на языке C++ со структурой таблицы лексем представлен на рисунке 3.5.

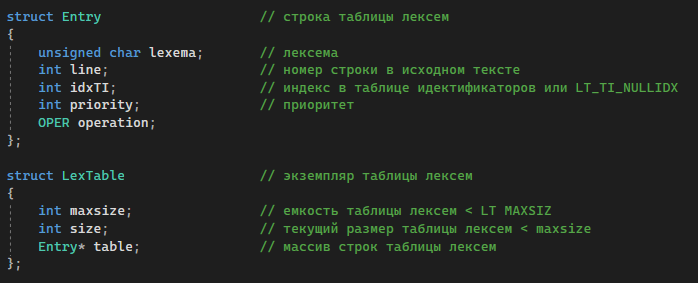


Рисунок 3.5 – Структура таблицы лексем

Код со структурой таблицы идентификаторов представлен на рисунке 3.6.

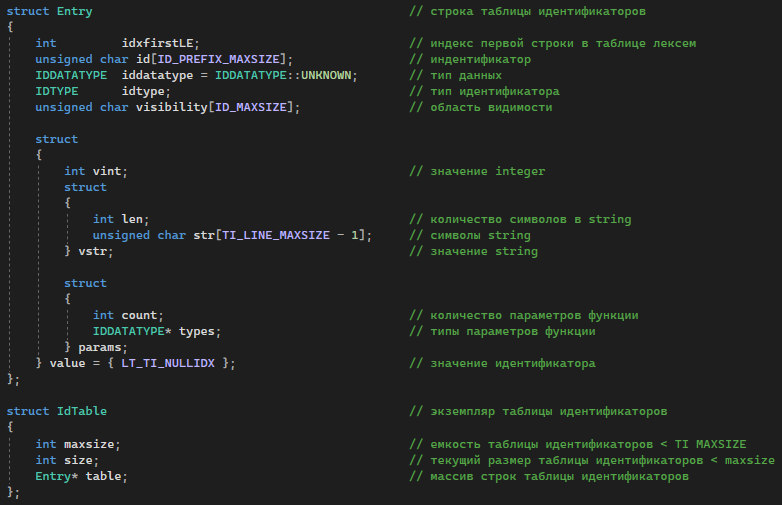


Рисунок 3.6– Структура таблицы идентификаторов

1. Структура и перечень сообщений лексического анализатора

Перечень сообщений лексического анализатора представлен на рисунке 3.7.

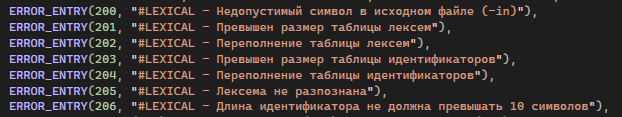


Рисунок 3.7– Сообщения лексического анализатора

1. 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 останавливает работу и далее сообщение выводится в файл протокола.

1. Параметры лексического анализатора и режимы его работы

Входными параметрами для лексического анализатора является исходный текст программы, написанный на языке RMV-2024, а также файл протокола, в который записываются выходные данные (таблица лексем и таблица идентификаторов).

1. Алгоритм лексического анализа

Лексический анализ основывается на работе конечных автоматов, разбирающих регулярные выражения. Алгоритм лексического анализа:

* из входного потока символов программы на исходном языке удаляются лишние пробелы и добавляется сепаратор для вычисления номера строки для каждой лексемы;
* формируется массив из слов языка;
* для каждого слова выполняется функция распознавания лексемы;
* при успешном распознавании информация о выделенной лексеме заносится в таблицу лексем и таблицу идентификаторов, и алгоритм возвращается к первому этапу;
* при неуспешном распознавании выдается сообщение об ошибке;
* формируется протокол работы.

1. Контрольный пример

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

4. Разработка синтаксического анализатора

1. Структура синтаксического анализатора

Синтаксический анализатор – это 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, полученные на этапе лексического анализа. Выходной информацией является дерево разбора.

Структура синтаксического анализатора представлена на рисунке 4.1.

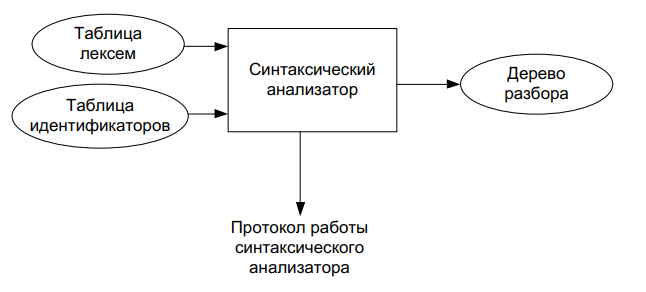


Рисунок 4.1 – Структура синтаксического анализатора

1. Контекстно свободная грамматика, описывающая синтаксис языка

В синтаксическом анализаторе транслятора языка RMV-2024 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.

Правила языка RMV-2024 реализованные на языке C++ представлены в приложении В.

Перечень правил, составляющих грамматику языка представлен в таблице 4.1.

Таблица 4.1 – Перечень правил, составляющих грамматику языка

|  |  |  |
| --- | --- | --- |
| Нетерминал | Цепочки правил | Описание |
| S | ftiFBS  siFUS  p{N}  ftiFB  siFU | Проверка правильности структуры программы |
| F | [P]  [] | Проверка наличия списка параметров функции |
| P | ti  ti,P | Проверка на ошибку в параметрах функции при ее объявлении |
| B | {Nr[I];}  {r[I];} | Проверка наличия тела функции |
| I | l  i | Проверка на недопустимое выражение (ожидается только литерал или идентификатор) |
| N | dti;N  dti;  dti=R;N  dti=E;N  dti=E;  i=R;N  i=E;N  i=E;  u[R]{X}N  u[R]{X}  w[R]{X}N  w[R]{X}  w[R]{X}!{X}N  w[R]{X}!{X}  %K;N  %K;  +K;N  +K;  qK;N  qK;  zK;N  zK;  o[I];N  o[I];  b[I];N  oK;  bK;  iK;N  iK; | Проверка на неверную конструкцию в теле функции |
| U | {N} | Проверка на ошибку в теле процедуры |

Таблица 4.1 (продолжение)

|  |  |  |
| --- | --- | --- |
| R | i  l  igi  igl  lgi  lgl | Проверка на ошибку в условном выражении |
| K | [W] [] | Проверка на ошибку в вызове функции |
| E | i  iM  l  lM  (E)  (E)M  iK  iKM  %K  %KM  +K  +KM  qK  qKM  zK  zKM | Проверка на ошибку в арифметическом выражении |
| W | i  l  i,W  l,W | Проверка на ошибку в параметрах вызываемой функции |
| M | vE  vEM | Проверка арифметических действий |
| X | dti;N  dti;  dti=E;N  dti=E;  dti=R;N  dti=R;  i=R;N  i=R;  i=E;N  i=E;  %K;N  %K;  +K;N  +K;  qK;N | Проверка на неверную конструкцию в теле цикла или условного выражения |

Таблица 4.1 (продолжение)

|  |  |  |
| --- | --- | --- |
|  | qK;  zK;N  zK;  o[I];N  oK;  b[I];N  bK;  iK;N  iK;  r[I];N  r[I]; |  |

1. 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. |
|  | Алфавит специальных магазинных символов | Алфавит магазинных символов содержит стартовый символ и маркер дна стека (представляет из себя символ $). |
|  | Функция переходов автомата | Функция, которая представляет из себя множество правил грамматики. |
|  | 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. |

1. Основные структуры данных

Основными структурами данных синтаксического анализатора являются структуры магазинного конечного автомата, выполняющего разбор исходной ленты, и структуры грамматики Грейбах, описывающей синтаксические правила языка RMV-2024. Данные структуры представлены в приложении Г.

1. Описание алгоритма синтаксического разбора

Принцип работы автомата, 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 выбирается другая цепочка;
6. Если в магазине встретился нетерминал, переход к пункту 4;
7. Если символ достиг дна стека и лента в этот момент пуста, то синтаксический анализ выполнен успешно. Иначе генерируется исключение.
8. Структура и перечень сообщений синтаксического анализатора

Перечень сообщений синтаксического анализатора представлен на рисунке 4.2.

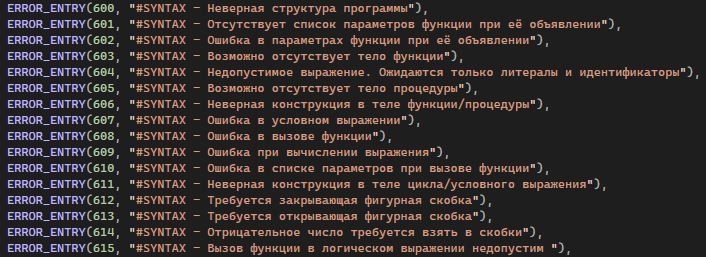


Рисунок 4.2– Сообщения синтаксического анализатора

1. Параметры синтаксического анализатора и режимы его работы

Входным параметром синтаксического анализатора является таблица лексем, полученная на этапе лексического анализа, протокол работы, а также правила контекстно-свободной грамматики в нормальной форме Грейбах.

Выходными параметрами являются трассировка прохода таблицы лексем и дерево разбора, которые записываются в файл протокола.

1. Принцип обработки ошибок

Обработка ошибок происходит следующим образом:

1. Синтаксический анализатор перебирает все правила и цепочки правила грамматики для нахождения подходящего соответствия с конструкцией, представленной в таблице лексем.
2. Если невозможно подобрать подходящую цепочку, то генерируется соответствующая ошибка.
3. Все ошибки записываются в общую структуру ошибок.
4. В случае нахождения ошибки, после всей процедуры трассировки в протокол будет выведено диагностическое сообщение.
5. Контрольный пример

Пример разбора синтаксическим анализатором исходного кода предоставлен в приложении Д в виде фрагмента трассировки и дерева разбора исходного кода.

5. Разработка семантического анализатора

1. Структура семантического анализатора

Семантический анализатор принимает на свой вход результаты работ лексического и синтаксического анализаторов и результат работы синтаксического анализатора и последовательно ищет необходимые ошибки. Некоторые проверки осуществляются в процессе лексического анализа. Общая структура семантического анализатора представлена на рисунке 5.1.

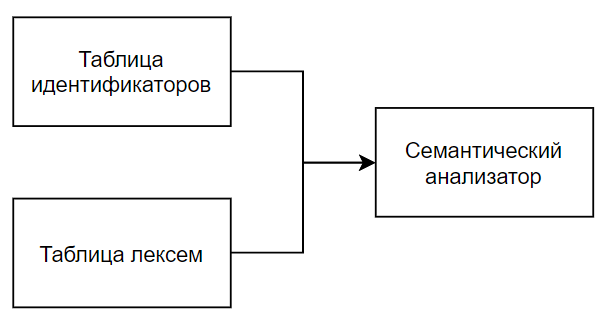


Рисунок 5.1 – Структура семантического анализатора

1. Функции семантического анализатора

Семантический анализатор выполняет проверку на основные правила языка (семантики языка), которые описаны в разделе 1.16. За семантический анализ отвечает функция AnalyzeSem. Ее входными параметрами является таблица лексем и таблица идентификаторов.

1. Структура и перечень сообщений семантического анализатора

Сообщения семантического анализатора, представлены на рисунке 5.2.

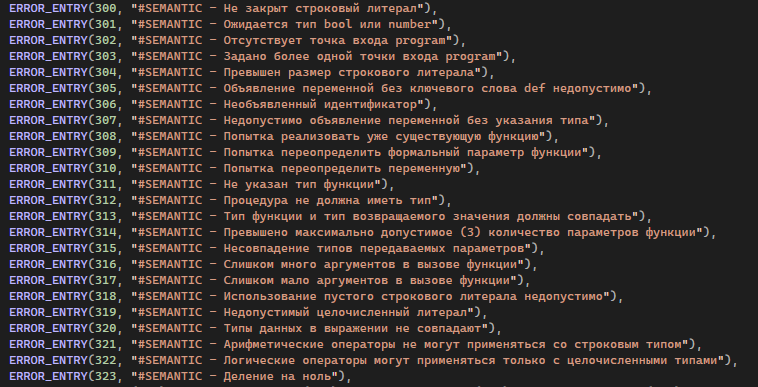


Рис. 5.2 – перечень сообщений семантического анализатора

1. Принцип обработки ошибок

Ошибки, возникающие во время трансляции программы, записываются в протокол, определённый входными параметрами. При возникновении ошибок осуществляется их протоколирование с указанием номера и диагностического сообщения, после чего семантический анализ останавливается.

1. Контрольный пример

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

Таблица 5.1 – Примеры семантических ошибок

|  |  |
| --- | --- |
| Исходный код с ошибкой | Генерируемое сообщение об ошибке |
| {  def number a = 5;  outln[a];  } | Ошибка 302: #SEMANTIC - Отсутствует точка входа program  Строка -1 позиция -1 |
| program  {  def line a = "Test;  outln[a];  } | Ошибка 300: #SEMANTIC - Не закрыт строковый литерал  Строка 3 позиция 15 |
| program  {  line a = "Test";  outln[a];  } | Ошибка 305: #SEMANTIC - Объявление переменной без ключевого слова def недопустимо  Строка 3 позиция 7 |
| program  {  def line a = 99999999999;  outln[a];  } | Ошибка 319: #SEMANTIC - Недопустимый целочисленный литерал  Строка 3 позиция 15 |
| program  {  def a = 67;  outln[a];  } | Ошибка 307: #SEMANTIC - Недопустимо объявление переменной без указания типа  Строка 3 позиция 6 |
| program  {  def line a = 67;  outln[a];  } | Ошибка 320: #SEMANTIC - Типы данных в выражении не совпадают  Строка 3 позиция -1 |
| function MyFunc [number a, number b]  {…} | Ошибка 311: #SEMANTIC - Не указан тип функции  Строка 1 позиция -1 |

Таблица 5.1 (продолжение)

|  |  |
| --- | --- |
| procedure number MyProc [number a]  {…} | Ошибка 312: #SEMANTIC - Процедура не должна иметь тип  Строка 1 позиция -1 |
| program  {  def line a = "Hello, " + "World!";  } | Ошибка 321: #SEMANTIC - Арифметические операторы не могут применяться со строковым типом  Строка 3 позиция -1 |
| function number MyFunc [number a]  {  def line lnA = "Hello";  give[lnA];  }  program  {…} | Ошибка 313: #SEMANTIC - Тип функции и тип возвращаемого значения должны совпадать  Строка 4 позиция -1 |
| program  {  def bool b = true < false;  } | Ошибка 322: #SEMANTIC - Логические операторы могут применяться только с целочисленными типами  Строка 3 позиция -1 |

6. Вычисление выражений

1. Выражения, допускаемые языком

В языке RMV-2024 допускается вычисление выражений целочисленного типа, также поддерживается вызов функций внутри арифметических выражений. В выражениях могут использоваться арифметические и логические операции. Операции и их приоритет представлен в таблице 6.1.

Таблица 6.1 – Операции и их приоритет

|  |  |  |
| --- | --- | --- |
| Тип операций | Операции | Приоритет |
| Логические операции | > – больше  < – меньше  == – равно  ^= – не равно  >= – больше или равно  <= – меньше или равно | -1  -1  -1  -1  -1  -1 |
| Арифметические операции | + – сложение  - – вычитание  \* – умножение  / – деление  % –остаток от деления | 2  2  3  3  3 |

Примеры выражений языка RMV-2024 представлены на рисунке 6.1.

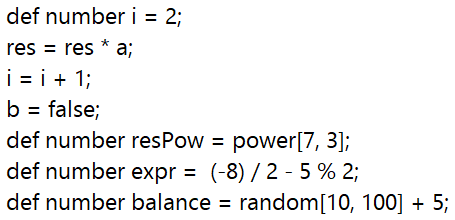


Рисунок 6.1 – Примеры выражений

1. Польская запись и принцип ее построения

Выражения в языке RMV-2024 преобразовываются к обратной польской записи.

Обратная польская запись — это форма записи математических выражений, в которой операторы расположены после своих операндов. Выражение в обратной польской нотации читается слева направо: операция выполняется над двумя операндами, непосредственно стоящими перед знаком этой операции.

Алгоритм построения:

* исходная строка: выражение;
* результирующая строка: польская запись;
* стек: пустой;
* результирующая строка: польская запись;
* исходная строка просматривается слева направо;
* операнды переносятся в результирующую строку в порядке их следования;
* операция записывается в стек, если стек пуст или в вершине стека лежит отрывающая скобка;
* операция выталкивает все операции с большим или равным приоритетом в результирующую строку;
* запятая не помещается в стек, если в стеке операции, то все выбираются в строку;
* отрывающая скобка помещается в стек;
* закрывающая скобка выталкивает все операции до открывающей скобки, после чего обе скобки уничтожаются;
* закрывающая квадратная скобка выталкивает все до открывающей и генерирует @ – специальный символ, в которого записывается информация о вызываемой функции, а в поле приоритета для данной лексемы записывается число параметров вызываемой функции;
* по концу разбора исходной строки все операции, оставшиеся в стеке, выталкиваются в результирующую строку.

В таблице 6.2 представлен пример преобразования выражения в обратную польскую запись.

Таблица 6.2 – Пример преобразования выражения в обратную польскую запись

|  |  |  |
| --- | --- | --- |
| Исходная строка | Результирующая строка | Стек |
| z[l,l]vl | - | - |
| [l,l]vl | z | - |
| l,l]vl | z | [ |
| ,l]vl | zl | [ |
| l]vl | zl | [ |
| ]vl | zll | [ |
| vl | zll@2 | - |
| l | zll@2 | v |
| - | zll@2l | v |
| - | zll@2lv | - |

1. Программная реализация обработки выражений

Программная реализация алгоритма преобразования выражений в обратную польскую запись основана на функциях Poliz и StartPoliz. Функция StartPoliz принимает как параметр таблицу лексем и таблицу идентификаторов и содержит цикл, в ходе которого перебираются все лексемы исходного кода. Если последовательность лексем соответствует началу выражения, вызывается функция Poliz, где и проводится преобразование выражений к польской нотации.

1. Контрольный пример

В приложении Е приведено представление промежуточного кода, отображающее результаты преобразования выражений в польский формат.

7. Генерация кода

1. Структура генератора кода

Генерация объектного кода — это перевод компилятором внутреннего представления исходной программы в цепочку символов выходного языка. На вход генератора подаются таблицы лексем и идентификаторов, на основе которых генерируется файл с ассемблерным кодом. Структура генератора кода представлена на рисунке 7.1.



Рисунок 7.1 Структура генератора кода

1. Представление типов данных в оперативной памяти

Элементы таблицы идентификаторов расположены сегментах .data и .const языка ассемблера. Соответствия между типами данных идентификаторов на языке RMV-2024 и на языке ассемблера приведены в таблице 7.1.

Таблица 7.1 – Соответствия типов идентификаторов

|  |  |  |
| --- | --- | --- |
| Тип идентификатора на языке RMV-2024 | Тип идентификатора на языке ассемблера | Пояснение |
| number | sword | Хранит целочисленный тип данных. |
| bool | sword | Хранит булевый тип данных (в виде целого числа) |
| line | dword | Хранит указатель на начало строки. Строка должна завешаться нулевым символом. |

1. Статическая библиотека

В языке RMV-2024 предусмотрена статическая библиотека. Статическая библиотека содержит функции, написанные на языке C++. Подключение библиотеки происходит с помощью includelib на этапе генерации кода. C помощью оператора EXTRN объявляются функции из библиотеки. Пример подключения библиотеки в исходном коде на языке ассемблера представлен на рисунке 7.2.

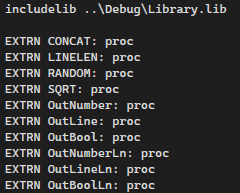


Рисунок 7.2 – Подключение статической библиотеки

1. Особенности алгоритма генерации кода

В языке RMV-2024 генерация кода строится на основе таблиц лексем и идентификаторов. В таблице 7.2 представлены прототипы функций, осуществляющих генерацию года, и их описание.

Таблица 7.2 – Прототипы функций, осуществляющих генерацию кода

|  |  |
| --- | --- |
| Прототипы функций | Описание |
| void Generation(LT::LexTable, IT::IdTable, wchar\_t) | Основная функция. Формирует поток выходного файла и вызывает другие генерирующие функции. |
| void Head(ofstream\*); | Функция, генерирующая заголовок ассемблерного файла (подключение библиотек, указание прототипов функций и т.д.). |
| void ConstSegment(IT::IdTable, ofstream\*); | Функция, генерирующая сегмент констант. |
| void DataSegment (LT::LexTable, IT::IdTable, ofstream\*) | Функция, генерирующая сегмент данных. |
| void CodeSegment (LT::LexTable, IT::IdTable, ofstream\*) | Функция, генерирующая сегмент кода. |

1. Входные параметры, управляющие генерацией кода

На вход генератору кода поступают таблицы лексем и идентификаторов. Результаты работы генератора кода выводятся в файл с расширением .asm.

1. Контрольный пример

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

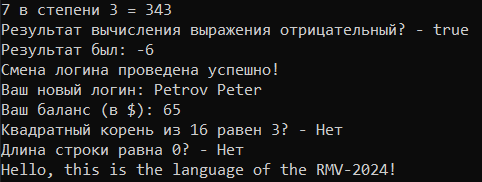


Рисунок 7.3 – Результат работы программы на языке RMV-2024

8. Тестирование транслятора

1. Общие положения

В процессе работы транслятора могут возникать ошибки на различных этапах трансляции: при анализе исходного текста программы, лексическом анализе, синтаксическом анализе и семантическом анализе. Транслятор отслеживает возникшие ошибки и записывает их в файл протокола, указывая идентификатор ошибки, сообщение, номер строки и позицию в исходном тексте программы. Обычно, после возникновения ошибки работа транслятора прекращается, так как ошибка на одном этапе может привести к сбоям на следующих этапах (за исключением синтаксического анализа).

1. Результаты тестирования

В таблице 8.1 приведены результаты тестов для разных этапов трансляции.

Таблица 8.1 – Результаты тестов

|  |  |
| --- | --- |
| Исходный код | Диагностическое сообщение |
| Проверка на допустимость символов | |
| p№rogram  {  } | Ошибка 200: #LEXICAL - Недопустимый символ в исходном файле (-in)  Строка 1 позиция 2 |
| Лексический анализ | |
| function number func! [number a, number b]  { …} | Ошибка 205: #LEXICAL - Лексема не распознана  Строка 1 позиция 17 |
| function number MyTallBehalfFunction [number a]  {…} | Ошибка 206: #LEXICAL - Длина идентификатора не должна превышать 10 символов  Строка 1 позиция 17 |
| Синтаксический анализ | |
| program def number a  {…} | Ошибка 600: строка 1, #SYNTAX - Неверная структура программы |
| function number myFunc  {  give[7];  }  program  {…} | Ошибка 601: строка 2, #SYNTAX - Отсутствует список параметров функции при её объявлении |
| function number myFun []  program  {…} | Ошибка 603: строка 3, #SYNTAX - Возможно отсутствует тело функции |

Таблица 8.1 (продолжение)

|  |  |
| --- | --- |
| function number myFunc []  {  give[1 \* 1];  }  program  {…} | Ошибка 604: строка 3, #SYNTAX - Недопустимое выражение. Ожидаются только литералы и идентификаторы |
| procedure myProc []  program  {…} | Ошибка 605: строка 2, #SYNTAX - Возможно отсутствует тело процедуры |
| function number myFunc []  {  program  {  def number a = 9;  }  } | Ошибка 606: строка 3, #SYNTAX - Неверная конструкция в теле функции |
| program  {  when[1 > 8 > 9]  {  def number a = 1;  }  } | Ошибка 607: строка 3, #SYNTAX - Ошибка в условном выражении |
| function number myFunc [number a]  {  a = a + 2;  give[a];  }  program  {  def number num = myFunc;  } | Ошибка 608: #SYNTAX - Ошибка в вызове функции  Строка 9 позиция -1 |
| function number myFunc[number a, number b]  {  def number res = a + b;  give[res];  }  program  {  def number res = myFunc[2,,3];  } | Ошибка 610: строка 8, #SYNTAX - Ошибка в списке параметров при вызове функции |

Таблица 8.1 (продолжение)

|  |  |
| --- | --- |
| program  {  def number num = 7 \* 12);  } | Ошибка 609: строка 3, #SYNTAX - Ошибка при вычислении выражения |
| def number i = 0;  when[i ^= 8]  {  cycle[i < 7]  {  i = i + 1;  }  }  } | Ошибка 611: строка 6, #SYNTAX - Неверная конструкция в теле цикла/условного выражения |
| program  def number a = 9;  } | Ошибка 613: #SYNTAX - Требуется открывающая фигурная скобка |
| program  {  def number a; | Ошибка 612: #SYNTAX - Требуется закрывающая фигурная скобка |
| …  def number a = 8 - -9; | Ошибка 614: #SYNTAX - Отрицательное число требуется взять в скобки  Строка 3 позиция -1 |
| Семантический анализ | |
| when["line"]  {  out[true];  } | Ошибка 301: #SEMANTIC - Ожидается тип bool или number  Строка 3 позиция -1 |
| def bool a = true; | Ошибка 302: #SEMANTIC - Отсутствует точка входа program |
| program  {  def line ln = "jdjdkkds ffhjaknks, haghvhj tuht … ADT GB FSHE FCyhh dgftsdtwg ytwytgwgj";  } | Ошибка 304: #SEMANTIC - Превышен размер строкового литерала |
| program  {  when[a == 1]  {}  } | Ошибка 306: #SEMANTIC - Необъявленный идентификатор  Строка 3 позиция -1 |
| program number def a  {…} | Ошибка 307: #SEMANTIC - Недопустимо объявление переменной без указания типа |

Таблица 8.1 (продолжение)

|  |  |
| --- | --- |
| function number MyFunc[number a]{…}  function number MyFunc[number a, number b]{…} | Ошибка 308: #SEMANTIC - Попытка реализовать уже существующую функцию  Строка 5 позиция -1 |
| function number MyFunc[number a, number b]  { def number a = 9; } | Ошибка 309: #SEMANTIC - Попытка переопределить формальный параметр функции |
| def line ln = "hello" + "world"; | Ошибка 321: #SEMANTIC - Арифметические операторы не могут применяться со строковым типом |
| program  {  def bool b = true > false;  } | Ошибка 322: #SEMANTIC - Логические операторы могут применяться только с целочисленными типами |
| def number a = 12 / 0; | Ошибка 323: #SEMANTIC - Деление на ноль |

# **Заключение**

В процессе выполнения курсовой работы был создан транслятор для языка программирования RMV-2024. Реализованы основные задачи курсового проекта, включая:

* разработку спецификации языка программирования;
* создание структуры транслятора;
* реализацию лексического анализатора;
* создание синтаксического анализатора;
* разработку семантического анализатора;
* реализацию генератора кода на языке ассемблера;
* проведение тестирования транслятора.

Итоговая версия языка RMV-2024 включает:

* три типа данных;
* поддержку операторов вывода;
* возможность вызова функций стандартной библиотеки;
* пять арифметических операторов для вычисления выражений;
* шесть логических операторов для работы с выражениями;
* поддержку функций, процедур, операторов циклов и условий.

Таким образом, в ходе выполнения курсового проекта были приобретены новые знания и навыки в области проектирования систем программирования и разработки программного обеспечения для таких систем.

# **Список использованных литературных источников**

1. Ахо, А. Компиляторы: принципы, технологии и инструменты / А. Ахо, Р. Сети, Дж. Ульман. – M.: Вильямс, 2003. – 768с.
2. Вирт, Н. Построение компиляторов/ Никлаус Вирт. - ДМК-Пресс – Москва, 2016. - 188 с.
3. Герберт, Ш. Справочник программиста по C/C++ / Шилдт Герберт. - 3-е изд. – Москва: Вильямс, 2003. - 429 с.
4. Ирвин К. Р. Язык ассемблера для процессоров Intel / К. Р. Ирвин. – M.: Вильямс, 2005. – 912с.

# **Приложение А**

Контрольный пример

$Пользовательские функции

function number power [number a, number b]

{

def number i = 2;

def number res = a;

when[b == 0]

{

give[1];

}

$Цикл: пока i меньше либо равно n

cycle [i <= b]

{

res = res \* a;

i = i + 1;

}

give [res];

}

$Процедура

procedure yourLogin [line login]

{

outln["Смена логина проведена успешно!"];

out["Ваш новый логин: "];

outln[login];

}

program

{

$Вызов пользовательских функций

out["7 в степени 3 = "];

def number resPow = power[7, 3];

outln[resPow];

out["Результат вычисления выражения отрицательный? - "];

def number expr = (-10) / 2 - 7 % 2;

def bool resNeg = expr <= 0;

outln[resNeg];

out["Результат был: "]; outln[expr];

$Вызов процедуры

def line login = "Petrov Peter";

yourLogin[login];

$Вызов функций стандартной библиотеки

out["Ваш баланс (в $): "];

def number balance = random[10, 100] + 5;

outln[balance];

out["Квадратный корень из 16 равен 3? - "];

def number resSqrt = sqrt[16];

when [resSqrt == 3]

{

outln["Да"];

}

otherwise

{

outln["Нет"];

}

out["Длина строки равна 0? - "];

def line lineA = "abcdef";

def number lengthA = linelen[lineA];

def bool resA = lengthA ^= 0;

when [resA]

{

outln["Нет"];

}

otherwise

{

outln["Да"];

}

def line resConcat = concat["Hello, ", "this is the language of the RMV-2024!"];

outln[resConcat];

}

Листинг А.1 – Исходный код контрольного примера

# **Приложение Б**

Результат работы лексического анализа:

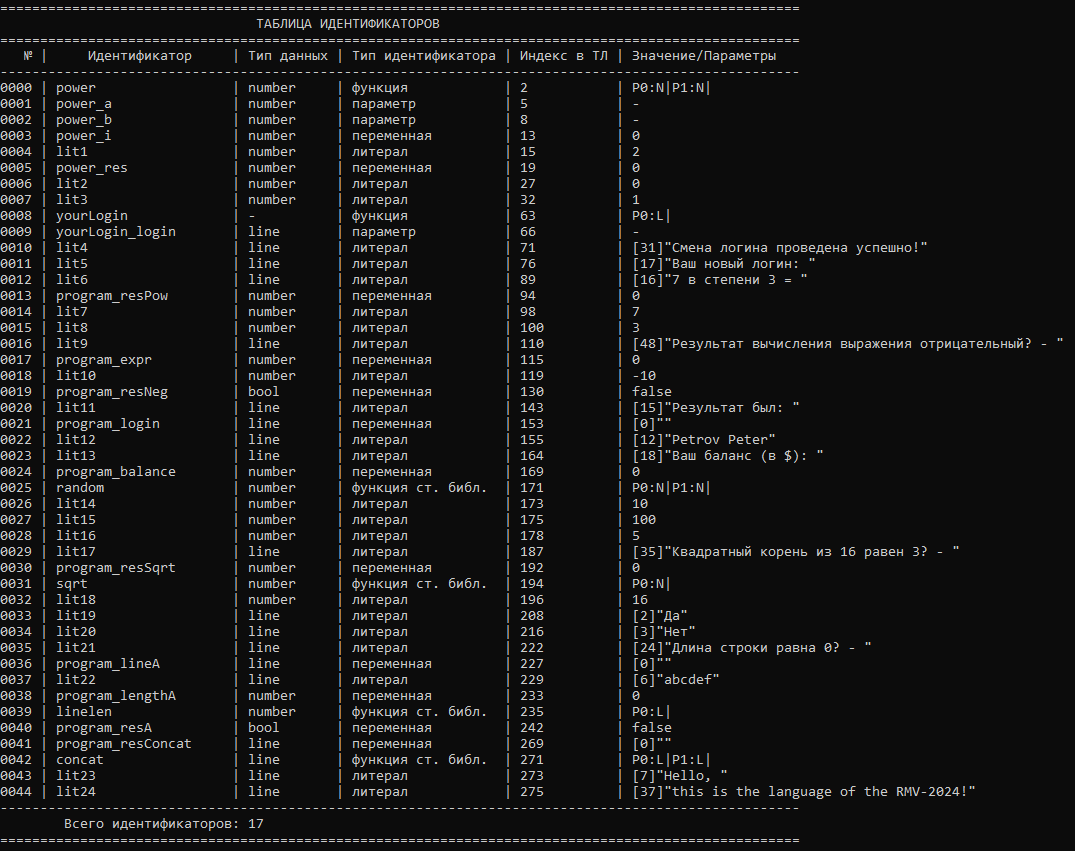
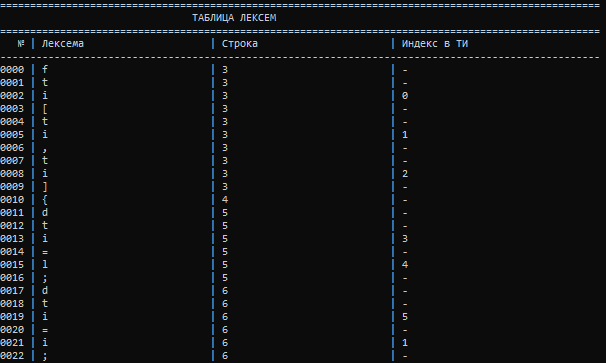


Рисунок Б.1 – Таблица идентификаторов

Начало таблицы лексем



Конец таблицы лексем

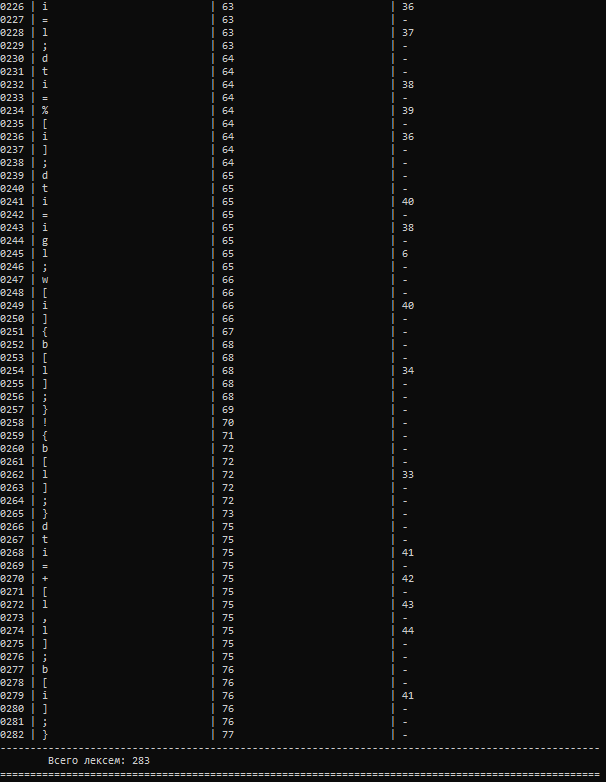


Рисунок Б.2 – Таблица лексем

# **Приложение В**

Грамматика языка

Greibach greibach(

NS('S'), TS('$'),

13,

Rule(

NS('S'), GRB\_ERROR\_SERIES + 0,

5,

Rule::Chain(6, TS('f'), TS('t'), TS('i'), NS('F'), NS('B'), NS('S')),

Rule::Chain(5, TS('s'), TS('i'), NS('F'), NS('U'), NS('S')),

Rule::Chain(4, TS('p'), TS('{'), NS('N'), TS('}')),

Rule::Chain(5, TS('f'), TS('t'), TS('i'), NS('F'), NS('B')),

Rule::Chain(4, TS('s'), TS('i'), NS('F'), NS('U'))

),

Rule(

NS('F'), GRB\_ERROR\_SERIES + 1,

2,

Rule::Chain(2, TS('['), TS(']')),

Rule::Chain(3, TS('['), NS('P'), TS(']'))

),

Rule(

NS('P'), GRB\_ERROR\_SERIES + 2,

2,

Rule::Chain(2, TS('t'), TS('i')),

Rule::Chain(4, TS('t'), TS('i'), TS(','), NS('P'))

),

Rule(

NS('B'), GRB\_ERROR\_SERIES + 3,

2,

Rule::Chain(8, TS('{'), NS('N'), TS('r'), TS('['), NS('I'), TS(']'), TS(';'), TS('}')),

Rule::Chain(7, TS('{'), TS('r'), TS('['), NS('I'), TS(']'), TS(';'), TS('}'))

),

Rule(

NS('I'), GRB\_ERROR\_SERIES + 4,

2,

Rule::Chain(1, TS('l')),

Rule::Chain(1, TS('i'))

),

Rule(

NS('U'), GRB\_ERROR\_SERIES + 5,

1,

Rule::Chain(3, TS('{'), NS('N'), TS('}'))

),

Rule(

NS('N'), GRB\_ERROR\_SERIES + 6,

33,

Rule::Chain(5, TS('d'), TS('t'), TS('i'), TS(';'), NS('N')),

Rule::Chain(7, TS('d'), TS('t'), TS('i'), TS('='), NS('R'), TS(';'), NS('N')),

Rule::Chain(7, TS('d'), TS('t'), TS('i'), TS('='), NS('E'), TS(';'), NS('N')),

Rule::Chain(5, TS('i'), TS('='), NS('R'), TS(';'), NS('N')),

Rule::Chain(5, TS('i'), TS('='), NS('E'), TS(';'), NS('N')),

Rule::Chain(8, TS('u'), TS('['), NS('R'), TS(']'), TS('{'), NS('X'), TS('}'), NS('N')),

Rule::Chain(8, TS('w'), TS('['), NS('R'), TS(']'), TS('{'), NS('X'), TS('}'), NS('N')),

Rule::Chain(12, TS('w'), TS('['), NS('R'), TS(']'), TS('{'), NS('X'), TS('}'), TS('!'), TS('{'), NS('X'), TS('}'), NS('N')),

Rule::Chain(6, TS('%'), NS('K'), TS(';'), NS('N')),

Rule::Chain(6, TS('+'), NS('K'), TS(';'), NS('N')),

Rule::Chain(6, TS('q'), NS('K'), TS(';'), NS('N')),

Rule::Chain(6, TS('z'), NS('K'), TS(';'), NS('N')),

Rule::Chain(6, TS('o'), TS('['), NS('I'), TS(']'), TS(';'), NS('N')),

Rule::Chain(6, TS('b'), TS('['), NS('I'), TS(']'), TS(';'), NS('N')),

Rule::Chain(4, TS('i'), NS('K'), TS(';'), NS('N')),

Rule::Chain(5, TS('r'), TS('['), NS('I'), TS(']'), TS(';')),

Rule::Chain(4, TS('d'), TS('t'), TS('i'), TS(';')),

Rule::Chain(6, TS('d'), TS('t'), TS('i'), TS('='), NS('E'), TS(';')),

Rule::Chain(6, TS('d'), TS('t'), TS('i'), TS('='), NS('R'), TS(';')),

Rule::Chain(4, TS('i'), TS('='), NS('R'), TS(';')),

Rule::Chain(4, TS('i'), TS('='), NS('E'), TS(';')),

Rule::Chain(7, TS('u'), TS('['), NS('R'), TS(']'), TS('{'), NS('X'), TS('}')),

Rule::Chain(7, TS('w'), TS('['), NS('R'), TS(']'), TS('{'), NS('X'), TS('}')),

Rule::Chain(11, TS('w'), TS('['), NS('R'), TS(']'), TS('{'), NS('X'), TS('}'), TS('!'), TS('{'), NS('X'), TS('}')),

Rule::Chain(3, TS('+'), NS('K'), TS(';')),

Rule::Chain(3, TS('%'), NS('K'), TS(';')),

Rule::Chain(3, TS('q'), NS('K'), TS(';')),

Rule::Chain(3, TS('z'), NS('K'), TS(';')),

Rule::Chain(5, TS('o'), TS('['), NS('I'), TS(']'), TS(';')),

Rule::Chain(3, TS('o'), NS('K'), TS(';')),

Rule::Chain(5, TS('b'), TS('['), NS('I'), TS(']'), TS(';')),

Rule::Chain(3, TS('b'), NS('K'), TS(';')),

Rule::Chain(3, TS('i'), NS('K'), TS(';'))

),

Rule(

NS('R'), GRB\_ERROR\_SERIES + 7,

6,

Rule::Chain(1, TS('i')),

Rule::Chain(1, TS('l')),

Rule::Chain(3, TS('i'), TS('g'), TS('i')),

Rule::Chain(3, TS('i'), TS('g'), TS('l')),

Rule::Chain(3, TS('l'), TS('g'), TS('i')),

Rule::Chain(3, TS('l'), TS('g'), TS('l'))

),

Rule(

NS('K'), GRB\_ERROR\_SERIES + 8,

2,

Rule::Chain(3, TS('['), NS('W'), TS(']')),

Rule::Chain(2, TS('['), TS(']'))

),

Rule(

NS('E'), GRB\_ERROR\_SERIES + 9,

16,

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('%'), NS('K')),

Rule::Chain(2, TS('+'), NS('K')),

Rule::Chain(2, TS('q'), NS('K')),

Rule::Chain(2, TS('z'), NS('K')),

Rule::Chain(2, TS('i'), NS('M')),

Rule::Chain(2, TS('l'), NS('M')),

Rule::Chain(4, TS('('), NS('E'), TS(')'), NS('M')),

Rule::Chain(3, TS('i'), NS('K'), NS('M')),

Rule::Chain(3, TS('%'), NS('K'), NS('M')),

Rule::Chain(3, TS('+'), NS('K'), NS('M')),

Rule::Chain(3, TS('q'), NS('K'), NS('M')),

Rule::Chain(3, TS('z'), NS('K'), NS('M'))

),

Rule(

NS('W'), GRB\_ERROR\_SERIES + 10,

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('M'), GRB\_ERROR\_SERIES + 9,

2,

Rule::Chain(2, TS('v'), NS('E')),

Rule::Chain(3, TS('v'), NS('E'), NS('M'))

),

Rule(

NS('X'), GRB\_ERROR\_SERIES + 11,

26,

Rule::Chain(5, TS('d'), TS('t'), TS('i'), TS(';'), NS('N')),

Rule::Chain(7, TS('d'), TS('t'), TS('i'), TS('='), NS('E'), TS(';'), NS('N')),

Rule::Chain(7, TS('d'), TS('t'), TS('i'), TS('='), NS('R'), TS(';'), NS('N')),

Rule::Chain(5, TS('i'), TS('='), NS('R'), TS(';'), NS('N')),

Rule::Chain(5, TS('i'), TS('='), NS('E'), TS(';'), NS('N')),

Rule::Chain(6, TS('%'), NS('K'), TS(';'), NS('N')),

Rule::Chain(6, TS('+'), NS('K'), TS(';'), NS('N')),

Rule::Chain(6, TS('q'), NS('K'), TS(';'), NS('N')),

Rule::Chain(6, TS('z'), NS('K'), TS(';'), NS('N')),

Rule::Chain(6, TS('o'), TS('['), NS('I'), TS(']'), TS(';'), NS('N')),

Rule::Chain(6, TS('b'), TS('['), NS('I'), TS(']'), TS(';'), NS('N')),

Rule::Chain(4, TS('i'), NS('K'), TS(';'), NS('N')),

Rule::Chain(6, TS('r'), TS('['), NS('I'), TS(']'), TS(';'), NS('N')),

Rule::Chain(4, TS('d'), TS('t'), TS('i'), TS(';')),

Rule::Chain(6, TS('d'), TS('t'), TS('i'), TS('='), NS('E'), TS(';')),

Rule::Chain(6, TS('d'), TS('t'), TS('i'), TS('='), NS('R'), TS(';')),

Rule::Chain(4, TS('i'), TS('='), NS('R'), TS(';')),

Rule::Chain(4, TS('i'), TS('='), NS('E'), TS(';')),

Rule::Chain(3, TS('+'), NS('K'), TS(';')),

Rule::Chain(3, TS('q'), NS('K'), TS(';')),

Rule::Chain(3, TS('z'), NS('K'), TS(';')),

Rule::Chain(3, TS('%'), NS('K'), TS(';')),

Rule::Chain(3, TS('o'), NS('K'), TS(';')),

Rule::Chain(3, TS('b'), NS('K'), TS(';')),

Rule::Chain(3, TS('i'), NS('K'), TS(';')),

Rule::Chain(5, TS('r'), TS('['), NS('I'), TS(']'), TS(';'))

)

);

Листинг В.1 – Правила грамматики языка

# **Приложение Г**

struct MfstState

{

short lenta\_position;

short nrule;

short nrulechain;

MFSTSTSTACK st;

MfstState();

MfstState(short pposition, MFSTSTSTACK pst, short pnrulechain);

MfstState(short pposition, MFSTSTSTACK pst, short pnrule, short pnrulechain);

};

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];

class my\_stack\_MfstState :public stack<MfstState>

{

public:

using stack<MfstState>::c;

};

GRBALPHABET\* lenta;

short lenta\_position;

short nrule;

short nrulechain;

short lenta\_size;

GRB::Greibach grebach;

LT::LexTable lex;

MFSTSTSTACK st;

my\_stack\_MfstState storestate;

Mfst();

Mfst(LT::LexTable& 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(Log::LOG log);

bool resetstate(Log::LOG log);

bool push\_chain(GRB::Rule::Chain chain);

RC\_STEP step(Log::LOG log);

bool start(Log::LOG log);

bool savediagnosis(RC\_STEP pprc\_step);

void printrules(Log::LOG log);

struct Deducation

{

short size;

short\* nrules;

short\* nrulechains;

Deducation()

{

size = 0;

nrules = 0;

nrulechains = 0;

};

} deducation;

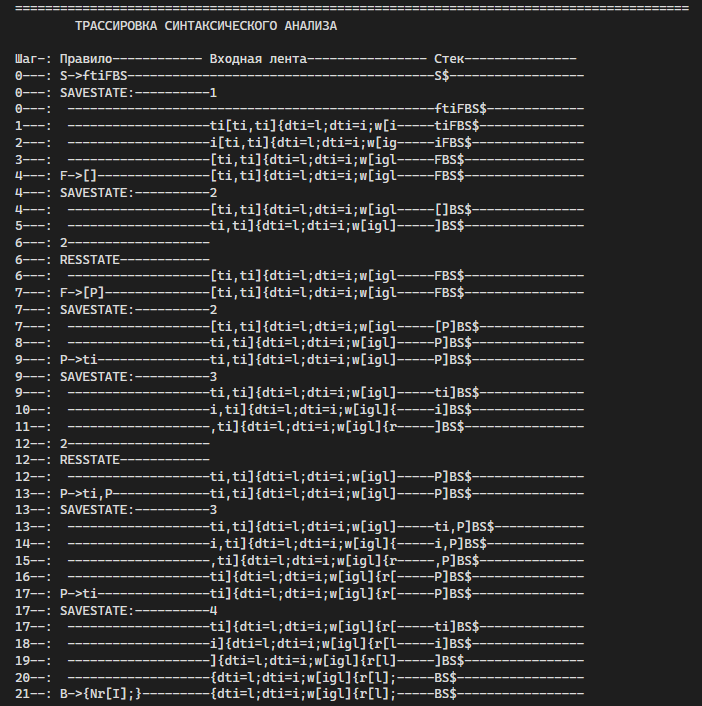
bool savededucation();

};

Листинг Г.1 – Структуры магазинного конечного автомата

# **Приложение Д**

Начало фрагмента трассировки синтаксического анализа



Конец фрагмента трассировки синтаксического анализа

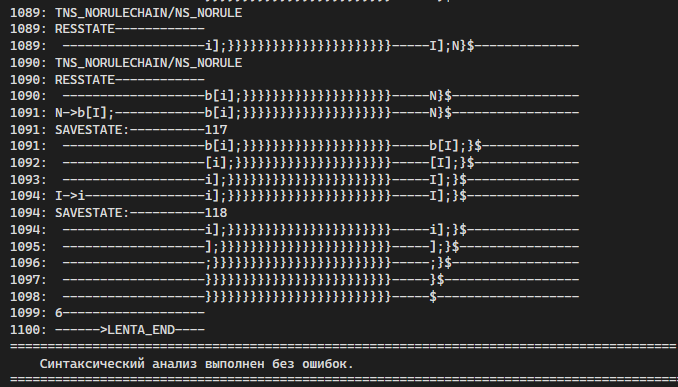
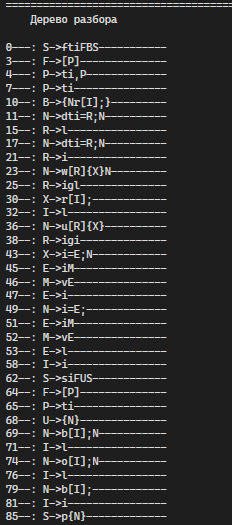


Рисунок Д.1 – Трассировка синтаксического анализа

Начало дерева разбора



Конец дерева разбора

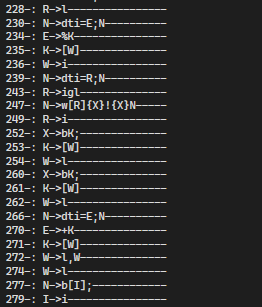


Рисунок Д.2 – Дерево разбора

# **Приложение Е**

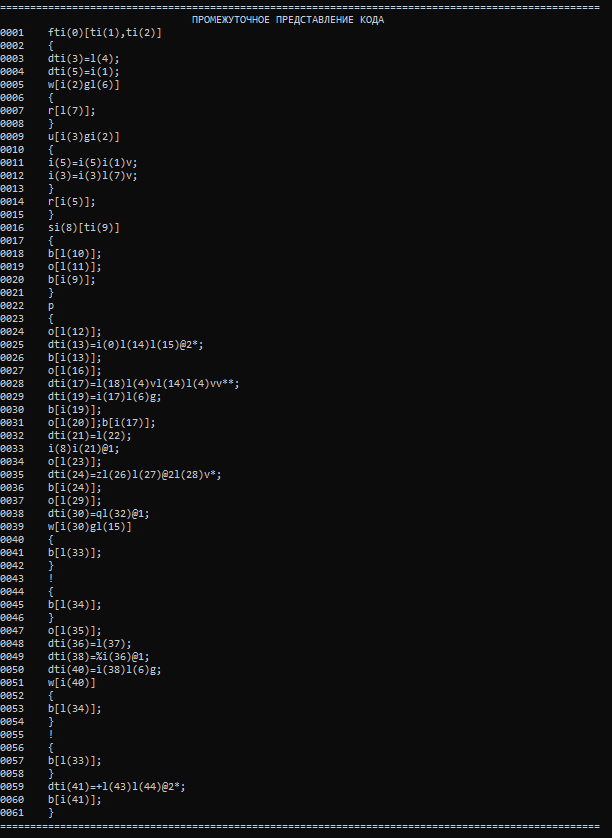


Рисунок Е.1 – Промежуточное представление кода

# **Приложение Ж**

Сгенерированный код ассемблера

.586P

.model flat, stdcall

includelib libucrt.lib

includelib kernel32.lib

includelib ..\Debug\Library.lib

ExitProcess PROTO :DWORD

EXTRN CONCAT: proc

EXTRN LINELEN: proc

EXTRN RANDOM: proc

EXTRN SQRT: proc

EXTRN OutNumber: proc

EXTRN OutLine: proc

EXTRN OutBool: proc

EXTRN OutNumberLn: proc

EXTRN OutLineLn: proc

EXTRN OutBoolLn: proc

.stack 4096

.const

null\_division BYTE "Exception: деление на ноль", 0

lit1 SWORD 2

lit2 SWORD 0

lit3 SWORD 1

lit4 BYTE "Смена логина проведена успешно!", 0

lit5 BYTE "Ваш новый логин: ", 0

lit6 BYTE "7 в степени 3 = ", 0

lit7 SWORD 7

lit8 SWORD 3

lit9 BYTE "Результат вычисления выражения отрицательный? - ", 0

lit10 SWORD -10

lit11 BYTE "Результат был: ", 0

lit12 BYTE "Petrov Peter", 0

lit13 BYTE "Ваш баланс (в $): ", 0

lit14 SWORD 10

lit15 SWORD 100

lit16 SWORD 5

lit17 BYTE "Квадратный корень из 16 равен 3? - ", 0

lit18 SWORD 16

lit19 BYTE "Да", 0

lit20 BYTE "Нет", 0

lit21 BYTE "Длина строки равна 0? - ", 0

lit22 BYTE "abcdef", 0

lit23 BYTE "Hello, ", 0

lit24 BYTE "this is the language of the RMV-2024!", 0

.data

buffer BYTE 256 dup(0)

power\_i SWORD 0

power\_res SWORD 0

program\_resPow SWORD 0

program\_expr SWORD 0

program\_resNeg SWORD 0

program\_login DWORD ?

program\_balance SWORD 0

program\_resSqrt SWORD 0

program\_lineA DWORD ?

program\_lengthA SWORD 0

program\_resA SWORD 0

program\_resConcat DWORD ?

.code

power PROC power\_a : SWORD, power\_b : SWORD

push lit1

pop power\_i

push power\_a

pop power\_res

mov ax, power\_b

cmp ax, lit2

je m0

jne m1

m0:

push 1

jmp local0

m1:

mov ax, power\_i

cmp ax, power\_b

jle cycle0

jmp cyclenext0

cycle0:

push power\_res

push power\_a

pop ax

pop bx

mul bx

push ax

pop power\_res

push power\_i

push lit3

pop ax

pop bx

add ax, bx

push ax

pop power\_i

mov ax, power\_i

cmp ax, power\_b

jle cycle0

cyclenext0:

push power\_res

jmp local0

local0:

pop eax

ret

power ENDP

yourLogin PROC yourLogin\_login : DWORD

push offset lit4

call OutLineLn

push offset lit5

call OutLine

push yourLogin\_login

call OutLineLn

local1:

pop eax

ret

ret

yourLogin ENDP

program PROC

push offset lit6

call OutLine

push lit7

push lit8

pop dx

pop dx

movsx eax, lit8

push eax

movsx eax, lit7

push eax

call power

push ax

pop program\_resPow

movsx eax, program\_resPow

push eax

call OutNumberLn

push offset lit9

call OutLine

push lit10

push lit1

pop bx

pop ax

cmp bx, 0

je nulldiv

cwd

idiv bx

push ax

push lit7

push lit1

pop bx

pop ax

cmp bx, 0

je nulldiv

cwd

idiv bx

push dx

pop bx

pop ax

sub ax, bx

push ax

pop program\_expr

push program\_expr

push lit2

pop bx

pop ax

cmp ax, bx

jle l0

jg l1

l0:

mov ax, 1

push ax

jmp endofexpr0

l1:

mov ax, 0

push ax

endofexpr0:

pop ax

cmp ax, 0

je l2

jne l3

l2:

mov ax, 0

push ax

jmp endofexpr1

l3:

mov ax, 1

push ax

endofexpr1:

pop program\_resNeg

movsx eax, program\_resNeg

push eax

call OutBoolLn

push offset lit11

call OutLine

movsx eax, program\_expr

push eax

call OutNumberLn

push offset lit12

pop program\_login

push program\_login

call yourLogin

push offset lit13

call OutLine

push lit14

push lit15

pop dx

pop dx

movsx eax, lit15

push eax

movsx eax, lit14

push eax

call RANDOM

push ax

push lit16

pop ax

pop bx

add ax, bx

push ax

pop program\_balance

movsx eax, program\_balance

push eax

call OutNumberLn

push offset lit17

call OutLine

push lit18

pop dx

movsx eax, lit18

push eax

call SQRT

push ax

pop program\_resSqrt

mov ax, program\_resSqrt

cmp ax, lit8

je m2

jne m3

m2:

push offset lit19

call OutLineLn

jmp e0

m3:

push offset lit20

call OutLineLn

e0:

push offset lit21

call OutLine

push offset lit22

pop program\_lineA

push program\_lineA

pop dx

push program\_lineA

call LINELEN

push ax

pop program\_lengthA

push program\_lengthA

push lit2

pop bx

pop ax

cmp ax, bx

jne l4

je l5

l4:

mov ax, 1

push ax

jmp endofexpr2

l5:

mov ax, 0

push ax

endofexpr2:

pop ax

cmp ax, 0

je l6

jne l7

l6:

mov ax, 0

push ax

jmp endofexpr3

l7:

mov ax, 1

push ax

endofexpr3:

pop program\_resA

mov ax, program\_resA

cmp ax, 0

jnz m4

jz m5

m4:

push offset lit20

call OutLineLn

jmp e1

m5:

push offset lit19

call OutLineLn

e1:

push offset lit23

push offset lit24

pop dx

pop dx

push offset lit24

push offset lit23

push offset buffer

call CONCAT

push eax

pop program\_resConcat

push program\_resConcat

call OutLineLn

theend:

push 0

call ExitProcess

nulldiv:

push offset null\_division

call OutLineLn

push -1

call ExitProcess

program ENDP

end program

Листинг Ж.1 –Код на языке ассемблера