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 LDV-2024»

Выполнил студент Литвинчук Дарья Валерьевна

(Ф.И.О.)

Руководитель проекта ст.преп. Наркевич Аделина Сергеевна

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

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

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

Консультанты

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

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

Нормоконтролер

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

Курсовой проект защищен с оценкой

Минск 2024

Содержание

[Введение 5](#_Toc185162495)

[1 Спецификация языка программирования 6](#_Toc185162496)

[1.1 Характеристика языка программирования 6](#_Toc185162497)

[1.2 Определение алфавита языка программирования 6](#_Toc185162498)

[1.3 Применяемые сепараторы 6](#_Toc185162499)

[1.4 Применяемые кодировки 7](#_Toc185162500)

[1.5 Типы данных 7](#_Toc185162501)

[1.6 Преобразование типов данных 8](#_Toc185162502)

[1.7 Идентификаторы 8](#_Toc185162503)

[1.8 Литералы 9](#_Toc185162504)

[1.9 Объявление данных 9](#_Toc185162505)

[1.10 Инициализация данных 10](#_Toc185162506)

[1.11 Инструкции языка 10](#_Toc185162507)

[1.12 Операции языка 11](#_Toc185162508)

[1.14 Конструкции языка 12](#_Toc185162509)

[1.15 Область видимости идентификаторов 13](#_Toc185162510)

[1.16 Семантические проверки 13](#_Toc185162511)

[1.18 Стандартная библиотека и ее состав 14](#_Toc185162512)

[1.19 Ввод и вывод данных 14](#_Toc185162513)

[1.20 Точка входа 14](#_Toc185162514)

[1.21 Препроцессор 14](#_Toc185162515)

[1.22 Соглашения о вызове 15](#_Toc185162516)

[1.23 Объектный код 15](#_Toc185162517)

[1.24 Классификация сообщений транслятора 15](#_Toc185162518)

[1.25 Контрольный пример 15](#_Toc185162519)

[2. Структура транслятора 16](#_Toc185162520)

[2.1 Компоненты транслятора, их назначение и принципы взаимодействия 16](#_Toc185162521)

[2.2 Перечень входных параметров транслятора 16](#_Toc185162522)

[2.3 Протоколы, формируемые транслятором и их содержимое 17](#_Toc185162523)

[3 Разработка лексического анализатора 18](#_Toc185162524)

[3.1 Структура лексического анализатора 18](#_Toc185162525)

[3.2 Контроль входных символов 19](#_Toc185162526)

[3.3 Удаление избыточных символов 19](#_Toc185162527)

[3.4 Перечень ключевых слов 19](#_Toc185162528)

[3.5. Основные структуры данных 20](#_Toc185162529)

[3.6 Структура и перечень сообщений лексического анализатора 22](#_Toc185162530)

[3.7 Принцип обработки ошибок 22](#_Toc185162531)

[3.8 Параметры лексического анализатора 22](#_Toc185162532)

[3.9 Алгоритм лексического анализа 22](#_Toc185162533)

[3.10 Контрольный пример 23](#_Toc185162534)

[4. Разработка синтаксического анализатора 24](#_Toc185162535)

[4.1 Структура синтаксического анализатора 24](#_Toc185162536)

[4.2 Контекстно-свободная грамматика, описывающая синтаксис языка 24](#_Toc185162537)

[4.3 Построение конечного магазинного автомата 27](#_Toc185162538)

[4.4 Основные структуры данных 28](#_Toc185162539)

[4.5 Описание алгоритма синтаксического разбора 31](#_Toc185162540)

[4.6 Параметры синтаксического анализатора 31](#_Toc185162541)

[4.7 Структура и перечень сообщений синтаксического анализатора 32](#_Toc185162542)

[4.8 Принцип обработки ошибок 32](#_Toc185162543)

[4.9 Контрольный пример 32](#_Toc185162544)

[5 Разработка семантического анализатора 33](#_Toc185162545)

[5.1 Структура семантического анализатора 33](#_Toc185162546)

[5.2 Функции семантического анализатора 33](#_Toc185162547)

[5.3 Структура и перечень сообщений семантического анализатора 33](#_Toc185162548)

[5.4 Принцип обработки ошибок 34](#_Toc185162549)

[5.5 Контрольный пример 34](#_Toc185162550)

[6. Вычисление выражений 35](#_Toc185162551)

[6.1 Выражения, допускаемые языком 35](#_Toc185162552)

[6.2 Польская запись и принцип её построения 35](#_Toc185162553)

[6.3 Программная реализация обработки выражений 36](#_Toc185162554)

[6.4 Контрольный пример 36](#_Toc185162555)

[7. Генерация кода 37](#_Toc185162556)

[7.1 Структура генератора кода 37](#_Toc185162557)

[7.2 Представление типов данных в оперативной памяти 37](#_Toc185162558)

[7.3 Статическая библиотека 38](#_Toc185162559)

[7.4 Особенности алгоритма генерации кода 38](#_Toc185162560)

[7.5 Параметры, управляющие генерацией кода 39](#_Toc185162561)

[7.6 Контрольный пример 39](#_Toc185162562)

[8. Тестирование транслятора 40](#_Toc185162563)

[8.1 Общие положения 40](#_Toc185162564)

[8.2 Результаты тестирования 40](#_Toc185162565)

[Заключение 43](#_Toc185162566)

[Список использованных источников 44](#_Toc185162567)

[Приложение А 45](#_Toc185162568)

[Приложение Б 47](#_Toc185162569)

[Приложение В 49](#_Toc185162570)

[Приложение Г 56](#_Toc185162571)

[Приложение Д 59](#_Toc185162572)

**Введение**

Целью курсового проекта является создание своего языка программирования LDV-2024 и реализация компилятора. Написание транслятора будет осуществляться на языке C++, при этом код на языке LDV-2024 будет транслироваться в язык ассемблера.

Компилятор LDV-2024 состоит из следующих частей:

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

На основе поставленной цели курсового проекта, были выявлены следующие задачи:

* Разработка спецификации языка LDV-2024;
* Разработка лексического анализатора;
* Разработка синтаксического анализатора;
* Разработка семантического анализатора;
* Разбор арифметических выражений;
* Разработка генератора кода;
* Тестирование транслятора.

# 1 Спецификация языка программирования

## **1.1 Характеристика языка программирования**

Язык программирования LDV-2024 является процедурным, универсальным строго типизированным, не объектно-ориентированным, компилируемым.

Поддерживает 2 типа данных: целочисленный (integer) и строковый (string). В стандартной библиотеке языка программирования доступны две функции: одна для лексикографического сравнения строк comp(string x, string y), а вторая для определения длины строки len(string str).

**1.2 Определение алфавита языка программирования**

При написании программы на языке LDV-2024 используется таблица символов Windows-1251.

Исходный код LDV-2024 может содержать символы-сепараторы, символы операций, символы латинского алфавита малого регистра, цифры десятичной системы счисления от 0 до 9, символы кириллицы [а…я], [А…Я] разрешены только в строковых литералах.

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

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

**1.3 Применяемые сепараторы**

Для того, чтобы разделить операции языка, необходимо использовать сепараторы. Используемые в языке программирования LDV-2024 сепараторы приведены в таблице 1.1.

Таблица 1.1. – Сепараторы

|  |  |
| --- | --- |
| Сепаратор | Назначение |
| пробел | Разделитель цепочек. Допускается везде кроме названий идентификаторов и ключевых слов |
| **{** … **}** | Блок функции или условной конструкции |
| **(** … **)** | Блок фактических или формальных параметров функции, а также приоритет арифметических операций |
| **,** | Разделитель параметров функций |
| **+ - \* / %** | Арифметические операции |
| **< > ! ~** | Операции сравнения |
| **;** | Разделитель программных конструкций |
| **=** | Оператор присваивания |

**1.4 Применяемые кодировки**

Для написания исходного кода программы на языке LDV-2024 используется кодировка Windows-1251. На рисунке 1.1 представлена таблица кодировки Windows-1251.



Рисунок 1.1 – Таблица кодировки Windows-1251

## **1.5 Типы данных**

Тип данных – это понятие, которое определяет набор значений, которые может принимать переменная или выражение, а также операции, которые могут быть выполнены над этими значениями. Тип данных определяет, как компьютер интерпретирует и обрабатывает информацию.

В языке LDV-2024 реализованы два типа данных: целочисленный и строковый. Описание типов приведено в таблице 1.2.

Таблица 1.2 – Типы данных языка LDV-2024

|  |  |
| --- | --- |
| Тип данных | Характеристика |
| integer | Целочисленный тип данных. Используется для работы с числовыми значениями. В памяти занимает 4 байта. Задание значения возможно в десятеричной и в шестнадцатеричной системах счисления. Инициализация по умолчанию: значение 0.  Поддерживаемые операции:  **+** (бинарный) – оператор сложения; |

Продолжение таблицы 1.2

|  |  |
| --- | --- |
|  | - (бинарный) – оператор вычитания;  **\*** (бинарный) – оператор умножения;  **/** (бинарный) – оператор деления;  **%** (бинарный) – оператор определения остатка от деления;  **=** (бинарный) – оператор присваивания.  В качестве условия условного оператора поддерживаются следующие логические операции:  **>** (бинарный) – оператор «больше»;  **<** (бинарный) – оператор «меньше»; |
| string | Строковый тип данных . Вся строка занимает n + 1 байт, где n – количество символов в строке и + 1 для символа конца строки.  Максимальная длина строки 255 символов.  Каждый символ занимает 1 байт, символы расположены в памяти друг за другом. Строка оканчивается символом конца строки.  Инициализация по умолчанию: \0.  Операции над данными строкового типа: присваивание строковому идентификатору значения другого строкового идентификатора, строкового литерала или значения строковой функции. |

**1.6 Преобразование типов данных**

В языке программирования LDV-2024 преобразование типов данных не поддерживаются.

**1.7 Идентификаторы**

Идентификаторы используются для наименования функций, параметров и переменных.

Имена идентификаторов могут содержать символы латинского алфавита нижнего регистра и цифры.

Максимальная длина идентификатора равна десяти символам.

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 и имена идентификаторов-переменных не должны совпадать с именами команд ассемблера.

Представление идентификатора в РБНФ:

<идентификатор>::= <строчная буква латинского алфавита>|< число>|< строчная буква латинского алфавита><идентификатор>

Пример правильных идентификаторов: a, b, num1

Пример неправильных идентификаторов: b$, e#h, Num, 1str

**1.8 Литералы**

Литерал – это запись в исходном коде программы, представляющая собой фиксированное значение.

С помощью литералов осуществляется инициализация переменных. Все литералы являются rvalue. Типы литералов языка LDV-2024 представлены в таблице 1.3.

Таблица 1.3 – Описание литералов

|  |  |
| --- | --- |
| Литералы | Пояснение |
| Целочисленные литералы в десятичном представлении | Может состоять из цифр [0-9]. Минимальное значение равно -2147483647, максимальное 2147483647. При выходе за пределы значение будет укорочено до максимального допустимого значения. |
| Строковые литералы | Состоит из символов, заключенных в " " (двойные кавычки). |
| Целочисленные литералы в шестнадцатеричном представлении | Может состоять из чисел [0-9], а также символов [A-F]. Начинается с суффикса 16x. Минимальное значение равно -16x7FFFFFFF, максимальное 16x7FFFFFFF. При выходе за пределы значение будет укорочено до максимального допустимого значения. |

Ограничения на строковые литералы языка LDV-2024: внутри литерала не допускается использование одинарных и двойных кавычек.

## **1.9 Объявление данных**

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

Пример объявления числового типа с инициализацией:

integer num1 = -1;

integer num2 = 16xFFF

Пример объявления переменной символьного типа с инициализацией:

string str1= "hello world"

Для объявления функций и процедур используется ключевое слово function, перед которым указывается тип функции. Далее обязателен список параметров и тело функции.

## **1.10 Инициализация данных**

Допустима инициализация при объявлении или присваивание значения уже объявленной ранее переменной.

При этом переменной будет присвоено значение литерала или идентификатора, стоящего справа от знака равенства.

Значения по умолчанию: значение 0 для беззнакового целочисленного типа данных, пустая строка ' ' для строкового типа данных.

Способы инициализации переменных языка программирования LDV-2024 представлены в таблице 1.4.

Таблица 1.4 – Способы инициализации переменных

|  |  |
| --- | --- |
| Вид инициализации | Примечание |
| <тип данных> <идентификатор>; | Автоматическая инициализация переменной integer нулем, string – пустой строкой. |
| <тип данных> <идентификатор> = <значение>; | Инициализация переменной с присваиванием значения. |

Объектами-инициализаторами могут быть только идентификаторы или литералы.

## **1.11 Инструкции языка**

Инструкции языка LDV-2024 представлены в таблице 1.5.

Таблица 1.5 – ИнструкцииязыкаLDV-2024

|  |  |
| --- | --- |
| Инструкция | Синтаксис |
| Объявление переменной | <тип данных> <идентификатор>; |
| Инициализация | <тип данных> <идентификатор> = <значение>;  Значение – инициализатор конкретного типа. Может быть только литералом или идентификатором |
| Возврат из функции или процедуры | Для функций, возвращающих значение:  return <идентификатор/литерал>;  Для процедур:  return; |
| Вывод данных | print <идентификатор/литерал>; |
| Вывод данных с переходом на новую строку | println <идентификатор/литерал>; |
| Вызов функции или процедуры | <идентификатор функции> (<список параметров>);  Список параметров может быть пустым. |
| Присваивание | <идентификатор> **=** <выражение>; |

Продолжение таблицы 1.5

|  |  |
| --- | --- |
|  | 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. Для строкового типа выражение может быть только идентификатором, литералом или вызовом функции, возвращающей значение строкового типа. |
| Условный оператор | if: <условие>  {…}  ^{…} |

**1.12 Операции языка**

В языке LDV-2024 предусмотрены следующие операции с данными. Приоритетность операции умножения выше приоритета операций сложения и вычитания. Операции языка представлены в таблице 1.6.

Таблица 1.6 – Операции языка LDV-2024

|  |  |
| --- | --- |
| Тип оператора | Оператор |
| Арифметические | **+ –** сложение  - – вычитание  \* – умножение  **/** – деление без остатка  **%** – остаток от деления |
| Строковые | **=**  – присваивание |
| Логические | **>** – больше  **<** – меньше  ~ – равно  ! – неравно |

Если у операций одинаковый приоритет, то первой будет выполнена операция, стоящая левее. С помощью круглых скобок может быть изменен приоритет операций.

**1.13 Выражения и их вычисления**

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 LDV-2024 организуется в виде функций пользователя и главной функции. Программные конструкции языка представлены в таблице 1.7.

Таблица 1.7 – Программные конструкции языка LDV-2024

|  |  |  |
| --- | --- | --- |
| Конструкция | | Реализация |
| Главная функция (main) | | main  {  …  } |
| Функция | | <тип данных> function <идентификатор> (<тип> <идентификатор>, ...)  {…  return <идентификатор|литерал>;  } |
| Процедура | void function <идентификатор> ( <тип> <идентификатор>, ...)  {…  return;  } | |
| Условный оператор | if: <идентификатор1> <оператор> <идентификатор2>  блок1{…}  ^блок2{…}  <идентификатор1>, <идентификатор2> - идентификаторы или литералы целочисленного типа (но не два литерала одновременно). <оператор> - один из операторов сравнения (>, <, ~, !), устанавливающий отношение между двумя операндами и организующий условие данной конструкции. При истинности условия выполняется код внутри блока1, иначе – код внутри блока блок2. | |

## **1.15 Область видимости идентификаторов**

Область видимости: сверху вниз. Переменные, объявленные в одной функции, недоступны в другой. Все объявления и операции с переменными происходят внутри блока где они были объявлены. 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.

**1.16 Семантические проверки**

В языке программирования LDV-2024 выполняются следующие семантические проверки:

* наличие функции main – 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 числовых литералов;
* правильность составленного условия условного оператора;
* запрет деления на ноль.

Семантические проверки необходимы для нахождения логических ошибок, обеспечения правильного смысла кода, а также для согласования разных частей программы.

**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 ее состав**

Язык программирования LDV-2024 предоставляет доступ к стандартной библиотеке, которая будет подключаться на этапе компоновки. Она написана на языке с++. Содержимое библиотеки и описание функций представлено в таблице 1.8.

Таблица 1.8 – Стандартная библиотека языка LDV-2024

|  |  |
| --- | --- |
| Функция | Описание |
| integer len(string str1); | Строковая функция, высчитывает длину строки. |
| integer comp(string str1, str2); | Сравнивает две строки и возвращает 1, если строки равны и 0, если строки не равны. |
| print |  |

## **1.19 Ввод и вывод данных**

Для вывода предусмотрены операторы print и println.

Таблица 1.9 – Дополнительные функции стандартной библиотеки

|  |  |
| --- | --- |
| Функция на языке С++ | Описание |
| void outn(int value) | Функция для вывода в стандартный поток значения целочисленного идентификатора/литерала. |
| void outnf(int value) | Функция для вывода в стандартный поток значения целочисленного идентификатора/литерала с переходом на новую строку. |
| void outw(char\* value) | Функция для вывода в стандартный поток значения строкового идентификатора/литерала. |
| void outwt(char\* value) | Функция для вывода в стандартный поток значения строкового идентификатора/литерала с переходом на новую строку. |

Для реализации вывода с переходом на новую строку были созданы копии функций с некоторым изменением.

## **1.20 Точка входа**

В языке LDV-2024 каждая программа должна содержать главную функцию (точку входа) main, с первой инструкции которой начнётся последовательное выполнение команд программы.

**1.21 Препроцессор**

В языке LDV-2024 препроцессор не предусматривается.

## **1.22 Соглашения о вызове**

В языке LDV-2024 используется стандартное соглашение о вызовах под названием stdcall – это значит, что в программе все параметры функции будут передаваться через стек справа налево и память будет освобождаться вызываемым кодом.

## **1.23 Объектный код**

Программа, исходный код которой написан на языке программирования LDV-2024, будет транслироваться в ассемблерный 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. Диапазоны кодов ошибок на разных фазах работы компилятора приведены в таблице 1.10.

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 |

**1.25 Контрольный пример**

Контрольный пример демонстрирует главные особенности языка LDV-2024: 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**

Транслятор – это программа, которая преобразует исходный код на одном языке программирования в исходный код на другом языке программирования.

На рисунке 2.1 представлена структура транслятора.

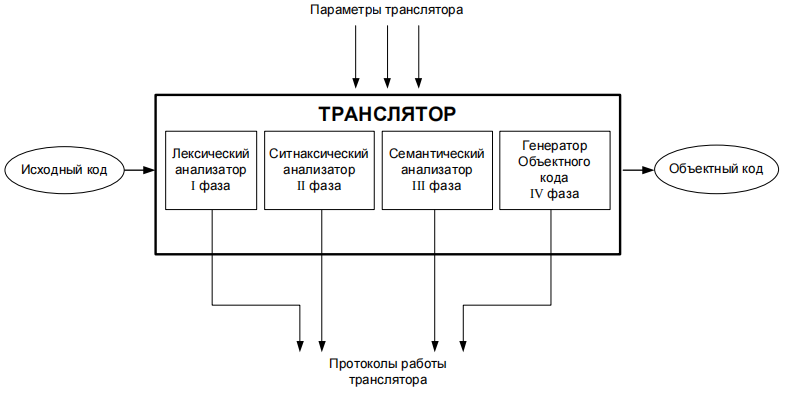


Рисунок 2.1 – Структура транслятора

Этапы, представленные на рисунке, выполняются последовательно. Каждый этап имеет свои входные и выходные данные, которые последовательно передаются на следующую фазу транслятора.

Первой фазой трансляции является лексический анализ. Он принимает на вход исходный код программы на языке высокого уровня. Задача лексического анализатора распознать лексемы и сформировать 2 таблицы: таблицу лексем и таблицу идентификаторов.

Синтаксический анализ представляет собой вторую часть работы транслятора. Входом для него является таблица лексем и таблица идентификаторов, полученные на предыдущем этапе. Задачей синтаксического анализатора является проверка кода на соответствие правилам грамматики и построение дерева разбора.

Третьей фазой является семантический анализ. Его задача – проверить выполнение семантических правил языка. На вход он принимает таблицу лексем, таблицу идентификаторов и дерево разбора.

Последний этап трансляции представляет собой генерацию кода. На вход генератора подаются таблица лексем и таблица идентификаторов, на основе которых генерируется файл с кодом на языке ассемблер.

**2.2 Перечень входных параметров транслятора**

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

Таблица 2.1 – Входные параметры транслятора языка LDV-2024

|  |  |  |
| --- | --- | --- |
| Входной параметр | Описание параметра | Значение по умолчанию |
| -in:<путь к in-файлу> | Файл с исходным кодом на языке LDV-2024, 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> | Файл с преобразованным кодом программы | Значение по умолчанию:  <имя in-файла>.out |

Таблица 2.1 содержит параметры, которые принимает транслятор языка LDV-2024 в качестве входных данных для создания файлов, содержащих результаты работы лексического, синтаксического и семантического анализаторов.

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

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 LDV-2024

|  |  |
| --- | --- |
| Формируемый протокол | Описание выходного протокола |
| Файл журнала, заданный параметром "-log:" | Файл с протоколом работы транслятора языка программирования LDV-2024. 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. |

При возникновении ошибки, компилятор прекратит свою работу, а информация об ошибке будет записана в log-файл. Генерация кода в ассемблер не будет выполнена в таком случае.

**3 Разработка лексического анализатора**

**3.1 Структура лексического анализатора**

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

Структурная схема лексического анализатора языка LDV-2024 представлена на рисунке 3.1.

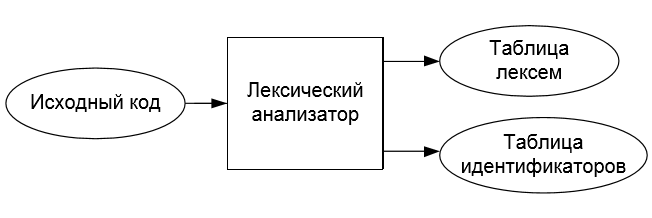


Рисунок 3.1 – Структурная схема лексического анализатора

Примеры лексических единиц включают идентификаторы, числа, символы операций и ключевые слова. Лексический анализатор преобразует исходный текст, заменяя лексические единицы их внутренним представлением - лексемами. Это создает промежуточное представление исходной программы. Каждой лексеме присваивается тип, и она записывается в таблицу идентификаторов, где хранится дополнительная информация о каждой лексеме.

Функции лексического анализатора:

* удаление «пустых» символов и комментариев. Если «пустые» символы

(пробелы, знаки табуляции и перехода на новую строку) и комментарии будут удалены лексическим анализатором, синтаксический анализатор никогда не столкнется с ними

* распознавание идентификаторов и ключевых слов;
* распознавание констант;
* распознавание разделителей и знаков операций.

Исходный код программы представлен в приложении А.

## **3.2 Контроль входных символов**

Для проверки входных символов на допустимость есть специальная таблица, которая повторяет таблицу Windows-1251. Символы делятся на несколько категорий: T – разрешенные символы, S – пробельные символы и знаки табуляции, Z – знаки операций.

Таблица контроля входных символов представлена на рисунке 3.2.

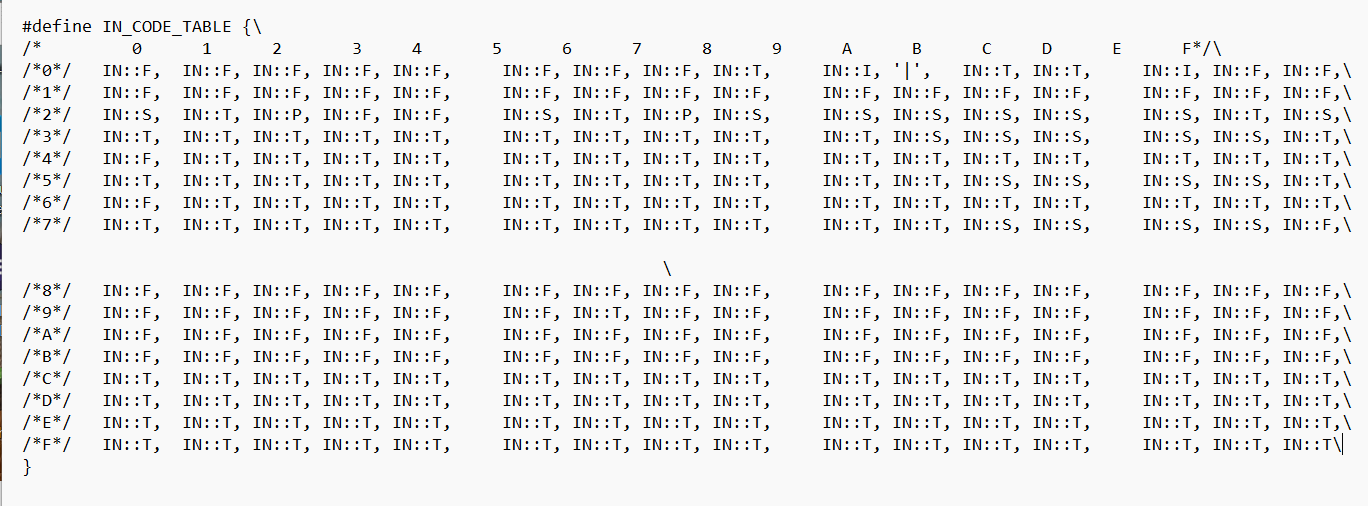


Рисунок 3.2 – Таблица проверки входных символов

## **3.3 Удаление избыточных символов**

Избыточный символ – это символ, удаление которого никак не влияет на исходный текст программы. В языке LDV-2024 избыточными символами являются пробельные символы, знаки табуляции. Избыточные символы удаляются на этапе разбиения исходного кода на лексемы.

Алгоритм удаления избыточных символов:

Пока не конец файла:

* читается символ;
* если символ является пробелом или знаком табуляции, то есть входит в категорию S:
* если перед или после символа стоят символы, которые относятся к категории Z:
* игнорируется текущий символ из категории S, так как он является избыточным;
* иначе данный символ категории S является сепаратором и его удаление приведет к некорректному анализу программы, а значит нужно его оставить.

## **3.4 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 3.1.

Таблица 3.1 – Соответствие токенов и сепараторов с лексемами

|  |  |  |
| --- | --- | --- |
| Токен | Лексема | Пояснение |
| integer, string | t | Названия типов данных языка. |
| Идентификатор | i | Длина идентификатора – 10 символов. |
| Литерал | l | Литерал любого доступного типа. |
| function | f | Объявление функции. |
| return | r | Выход из функции/процедуры. |
| main | m | Главная функция. |
| if: | w | Указывает начало условного оператора. |
| ^ | ^ | Разделение конструкций в условном операторе |
| ; | ; | Разделение выражений |
| , | , | Разделение параметров функций |
| { | { | Начало блока/тела функции |
| } | } | Закрытие блока/тела функции |
| ( | ( | Передача параметров в функцию, приоритет операций |
| ) | ) | Закрытие блока для передачи параметров, приоритет операций |
| =  +  -  \*  /  % | =  +  -  \*  /  % | Знак присваивания  Знаки операций |
| >  <  !  ~ | >  <  !  ~ | 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.

## **3.5. Основные структуры данных**

Основными структурами данных лексического анализатора являются таблица лексем и таблица идентификаторов.

Структура с именем LexTable представляет собой экземпляр таблицы лексем, в которой maxsize – максимальный размер таблицы, size – текущий размер таблицы, table – указатель на строку таблицы лексем. Данная таблица предназначена для хранения записей обо всех лексемах, которые встречаются в исходном коде программы на языке программирования LDV-2024.

Структура с именем Entry представляет экземпляр строки таблицы лексем, в которой lexema хранит символьное обозначение лексемы, sn – номер строки в исходном коде, в которой располагается данная лексема, idTI – номер строки в таблице идентификаторов.

Перечисление с именем IDDATATYPE используется для задания каждому идентификатору типа данных, который он может хранить. Для беззнакового целочисленного INT = 1, для строкового STR = 2.

Перечисление с именем IDTYPE используется для задания каждому идентификатору его типа. Для переменной V = 1, для функции F = 2, для параметра функции P = 3, для литерала L = 4.

Структура с именем IdTable представляет собой экземпляр таблицы идентификаторов, в которой maxsize – максимальный размер таблицы, size – текущий размер таблицы, table – указатель на строку таблицы идентификаторов. Данная таблица предназначена для хранения записей обо всех идентификаторах, которые встречаются в исходном коде программы на языке программирования LDV-2024.

Структура с именем Entry представляет экземпляр строки таблицы идентификаторов, в которой idxfirstLE – индекс первого вхождения идентификатора в таблицу лексем, id – идентификатор, iddatatype – тип данных, который может хранить данный идентификатор, idtype – тип самого идентификатора, scope\_name – имя области видимости, в которой находится данный идентификатор и в которой он будет доступен, value – объединение, которое позволяет хранить значение целочисленного, вещественного и логического идентификатора в переменной vint и структура vstr, которая нужна для хранения значения строкового идентификатора, в которой есть поля len для хранения длины строки и str для хранения самой строки. Также присутствуют несколько конструкторов: c параметрами и по умолчанию.

Код C++ со структурой таблицы идентификаторов представлен в Листинге 3.1.

struct Entry // строка таблицы идентификаторов

{

int idxfirstLE;

unsigned char id[ID\_MAXSIZE];

IDDATATYPE iddatatype = NUL;

IDTYPE idtype;

int parm = 0;

int nums = 0; //0 - 10 СС 1 - 16 СС

union

{

int vint;

struct

{} vstr;

} value;};

Листинг 3.1 – Структура таблицы лексем

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

В случае возникновения ошибки на стадии лексического анализа формируется экземпляр ошибки в следующем формате: код ошибки в соответствии с таблицей ошибок, текст сообщения, номер строки в исходном коде, позиция в строке.

## **3.7 Принцип обработки ошибок**

Если была обнаружена ошибка при выполнении лексического анализа, то формируется сообщение об ошибке и выводится на консоль, а также записывается в файл протокола работы, заданный параметром –log: <имя файла>.log. Транслятор прекращает свою работу. Перечень сообщений представлен в таблице 3.2.

Таблица 3.2 – Перечень сообщений ошибок лексического анализатора

|  |  |
| --- | --- |
| Номер ошибки | Текст сообщения |
| 200 | Неинициализированная переменная |
| 201 | Неверное имя идентификатора |
| 202 | Превышен допустимый размер литерала |

## **3.8 Параметры лексического анализатора**

На вход лексического анализатора подается исходный текст программы на языке программирования LDV-2024, в которым заранее были удалены избыточные символы.

## **3.9 Алгоритм лексического анализа**

Алгоритм лексического анализа описан ниже.

1. Читается посимвольно исходный код программы. Если текущий символ является концом строки, то переходим к пункту 8;
2. Если текущий символ является сепаратором, то он записывается в таблицу лексем, иначе записывается в буфер, пока не встретится сепаратор;
3. После этого строка, записанная в буфере, передается на распознавание конечным автоматам. При успешном разборе автомат вернет лексическому анализатору лексему, соответствующую данному ключевому слову, которое записывается в таблицу лексем. Если текущая лексема представляет собой идентификатор или литерал, то переход к пункту 4. В противном случае переход к пункту 1;
4. Если предыдущие лексемы являются ключевыми словами для обозначения типа данных, то текущему идентификатору или литералу присваивается данный тип;
5. Если текущая лексема идентификатор и происходит его объявление, то осуществляется поиск по таблице идентификаторов. В случае, если такой идентификатор уже существует в данной области видимости, то выводится ошибка и транслятор прекращает свою работу. В противном случае в таблицу идентификаторов заносится запись о текущем идентификаторе;
6. Если текущая лексема идентификатор и перед ним не указан тип данных, то есть предполагается, что он был объявлен ранее, то осуществляется поиск по таблице идентификаторов. В случае, если такого идентификатора не существует в данной области видимости, то поиск осуществляется в области видимости, предшествующей данной, пока не дойдет до глобальной. Если запись в таблице идентификаторов не была найдена, то выводится соответствующая ошибка и транслятор завершает свою работу. В противном случае запись в таблицу идентификаторов производится не будет, но будет сделана запись в таблицу лексем с ссылкой на найденную запись в таблице идентификаторов;
7. Если текущая лексема является литералом, то выявляется его тип и значение. Далее в таблицу идентификаторов помещается запись о текущем литерале. В таблицу лексем заносится данная лексема с ссылкой на таблицу идентификаторов. Переход в пункт 1;
8. Если идентификатор является функцией, то запись о ней будет помещена в таблицу идентификаторов с соответствующими типом возвращаемого значений. Последующие идентификаторы в круглых скобках будут записаны как параметры. Переход в пункт 1;
9. Если не дошли до конца исходного текста, то переход в пункт 1.

**3.10 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 анализ является второй фазой транслятора языка LDV-2024 и выполняется сразу после завершения фазы лексического анализатора. Он предназначен для распознавания заранее заданных синтаксических правил. Описание структуры синтаксического анализатора языка представлено на рисунке 4.1.

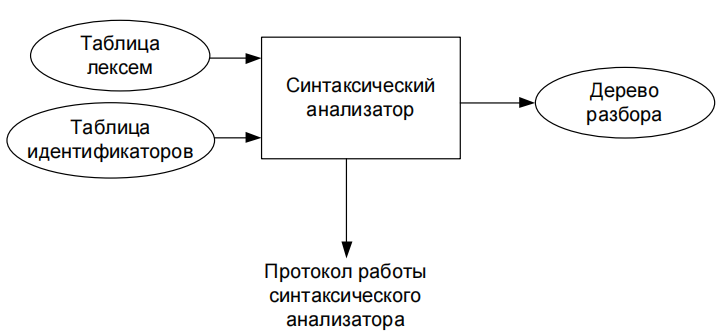


Рисунок 4.1 – Структура синтаксического анализатора

Входными данными для синтаксического анализатора являются таблица лексем и идентификаторов. Результатом работы синтаксического анализатора является дерево разбора, которое представляет собой иерархическую структуру программы, отражающую их вложенность и последовательность операций.

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

В синтаксическом анализаторе транслятора языка LDV-2024 используется контекстно-свободная грамматика , где

T – множество терминальных символов,

N – множество нетерминальных символов,

P – множество правил языка,

S – начальный символ грамматики, являющийся нетерминалом.

Данная грамматика G имеет нормальную форму Грейбах, так как она не является леворекурсивной (не содержит леворекурсивных правил) и все правила из конечного множества P имеют вид:

* A → aα, где a ∈ T, α ∈ N\*;
* S → λ, где S ∈ N – начальный символ, если есть такое правило, то S не должен встречаться в правой части правил.

Перед приведением грамматики к нормальной форме Грейбах необходимо:

* удалить все бесплодные символы;
* удалить все недостижимые символы из грамматики;
* удалить все лямбда-правила из грамматики;
* удалить все цепные правила.

Все синтаксические правила в нормальной форме Грейбах приведены в таблице 4.1.

Таблица 4.1 – Таблица правил переходов нетерминальных символов

|  |  |  |
| --- | --- | --- |
| Символ | Правила | Какие правила порождает |
| S | S->m{N}  S->tfi()TS  S->tfi(F)TS  S->nfi()TS  S->nfi(F)TS  S->m{N} | Стартовые правила, описывающее общую структуру программы |
| T | T->{N}  T->{Nr;}  T->{NrE;}  T->{rE;} | Правила для тела функций |
| N | N->ti;N  N->i(W);N  N->i(W);  N->i();N  N->i();  N->ti=E;N  N->ti=E;  N->i=E;  N->i=E;N  N->pi;N  N->zi;N  N->pl;N  N->zl;N  N->pE;N  N->zE;N  N->pE;  N->zE;  N->ti;  N->pi;  N->zi;  N->pl;  N->zl;  N->wI|X  N->wI|XN | Правила набора операций |
| B | B->ti=E;  B->i=E;  B->pE;  B->zE;  B->ti;  B->pi;  B->zi;  B->pl;  B->wI|X  B->wI|XN | Правила выражений |
| X | X->{N}^{N}  X->{N} | Правила для условного оператора |
| O | O->{N} | Правила для цикла |
| E | E->i;  E->l;  E->(E)  E->i(W)  E->i()  E->iM  E->lM  E->(E)M  E->i(W)M  E->i()M | Правила для выражений |
| F | F->ti  F->ti,F | Правила для передаваемых параметров |
| W | W->i  W->l  W->I,W  W->l,W | Правила для вызываемых параметров |
| M | M->vE  M->vEM | Правила для арифметических операторов |
| I | I->:EqE  I->:EqE&I | Правила для тела условного оператора |

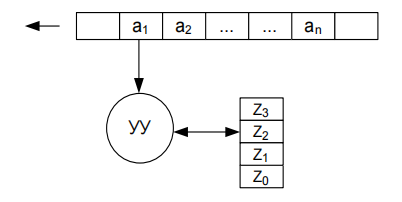
В таблице 4.1 представлено описание нетерминальных символов и соответствующих правил переходов в контекстно-свободной грамматике языка LDV-2024.

**4.3 Построение конечного магазинного автомата**

Конечный автомат с магазинной памятью – это семерка M = {Q, V, Z, δ, q0, z0, F}, в которой

* Q – множество возможных состояний;
* V – алфавит входных символов;
* Z – специальный алфавит магазинных символов;
* δ – функция переходов автомата Q × (V ∪{λ})× Z → P(Q × Z \*), где P(Q × Z \*) – множество подмножеств Q × Z \*;
* q0 ∈ Q – начальное состояние автомата;
* z0 ∈ Z – начальное состояние магазина (маркер дна);
* F ⊆ Q – множество конечных состояний.

Схема работы конечного автомата с магазинной памятью представлена на рисунке 4.2.

Рисунок 4.2 – Схема работы конечного автомата с магазинной памятью

Во время работы автомата на каждом шаге возможны 3 случая:

* функция δ(q, a, z) определена – осуществляется переход в новое состояние;
* функция δ(q, a, z) не определена, но определена δ(q, λ, z) – осуществляется переход в новое состояние (лента не продвигается);
* функция δ(q, a, z) и δ(q, λ, z) не определены – дальнейшая работа автомата не возможна (цепочка не разобрана).

**4.4 Основные структуры данных**

Структуры синтаксического анализатора и их описание представлены в таблице 4.2.

Таблица 4.2 – Структуры синтаксического анализатора

|  |  |
| --- | --- |
| Структура | Характеристика |
| Greibach | Структура: представление грамматики. Все символы (алфавит) грамматики представляются в формате **GRBALPHABET** (short). Причем терминалы – положительные значения, нетерминалы – отрицательные значения.  Структура включает:  - множество правил: переменная rules (типа структура **Rule**);  - количество правил: переменная **size** (short);  - стартовый символ грамматики: **startN** (**GRBALPHABET**);  - служебный символ (дно стека и последняя лексема таблице лексем):  **stbottomT** (**GRBALPHABET**);  - два конструктора;  - методы **getRule**:  1) позволяет получить номер правила или -1 (к точке возврата) и правило (второй параметр типа Rule) по левому символу правила (первый параметр);  2) позволяет получить правило (возвращает к точке вызова параметр типа Rule) по его номеру |

Продолжение таблицы 4.2 – Структуры синтаксического анализатора

|  |  |
| --- | --- |
| Структура | Характеристика |
| Rule | Структура: представление одного правила, имеющего вид: A→xxx|yyy|….  Структура включает:  - нетерминал – левый символ правила: **nn** (**GRBALPHABET**);  - идентификатор ошибки, связанной с правилом: **iderror** (int) – код ошибки в подсистеме **Error**;  - количество цепочек в правой стороне правила: **size** (short);  - цепочки-правые стороны правила: **chains** (типа **Rule::Chain**);  - два конструктора;  - метод **getCRule**: позволяет получить правило в виде строки вида N-> цепочка (в символьном ASCII-виде, для отображения);  - метод **getNextChain**: позволяет найти следующую за номером (3й параметр **j** типа short) цепочку (параметр **pchain** типа **Rule::Chain**) и ее номер (к точке возврата типа short |
| Rule::Chain | Структура: представление цепочки – правой стороны правила.  Структура включает:  - размер цепочки: **size** (short) в символах;  - цепочка: **nt** (**GRBALPHABET**);  - два конструктора;  - метод **getCChain**: позволяет получить строку-цепочку в  символьном виде для отображения;  - методы **T** и **N**: преобразовывают ASCI-символы в **GRBALPHABET-**символы (терминалы и нетерминалы);  - методы **isT** и **isN**: проверяют является **GRBALPHABET-**символ терминалом или нетерминалом;  - метод **aplphabet\_to\_char**: преобразует заданный (параметр) **GRBALPHABET-**символ в ASCII-символ |
| MfstState | Структура: для сохранения состояния автомата;  сохранять состояние автомата необходимо для того, чтобы иметь возможность к этому состоянию вернуться и осуществить альтернативный вариант синтаксического разбора (в силу недетерминированности автомата).  Структура включает:  - текущую позицию на входной ленте автомата: **lenta\_position** (short);  - номер текущей цепочки, текущего правила: **nrulechain** (short);  - стек автомата с содержимым на момент сохранения **st**  (**MFSTSTACK**);  - два конструктора. |

Продолжение таблицы 4.2 – Структуры синтаксического анализатора

|  |  |
| --- | --- |
| Структура | Характеристика |
| Mfst | Структура: представление магазинного конечного автомата.  Структура включает:  - перечисление, содержащее возможные коды возврата метода step: **RC\_STEP**;  - массив структур для строк диагностики: **diagnosis (MstDiagnosis**, описание ниже**)**;  - входную ленту: **lenta (GRBALPHABET\*)**;  - текущая позиция на входной ленте: **lenta\_position** (short);  - номер текущего правила грамматики: **nrule** (short);  - номер текущей цепочки текущего правила грамматики: **nrulechain** (short);  - количество символов на ленте: **lenta\_size** (short);  - грамматика языка: **grebach (**GRB::Greibach);  - результат, предварительно выполненного лексического анализа (таблицы лексем и идентификаторов): **lex** (**LEX::LEX**);  - стек автомата: **st** (**MFSSTATACK**);  - стек для хранения состояний (структур **MfstState**) автомата: **storestate** (std::stack< **MfstState**>);  - два конструктора;  - функция **getCSt:** принимает один параметр – буфер; заполняет буфер содержимым стека для отображения, в конце 0x00; возвращает к точке вызова указатель на буфер;  - функция **getCLenta:** заполняет буфер (первый параметр) содержимым ленты с заданной позиции (второй параметр) заданное количество символов (третий параметр) в формате ASCII-строки для отображения, в конце строки 0x00; возвращает к точке вызова указатель на строки буфер;  - функция **getDiagnosis:** по заданному номеру (первый параметр) строки диагностики записывает строку в буфер (второй параметр) в формате ASCII-строки для отображения и возвращает указатель на буфер;  - функция **savestate:** сохраняет текущее состояние автомата в **storestate,** всегда возвращает **true**;  - функция **reststate:** восстанавливает последнее сохраненное состояние автомата из **storestate**, возвращает **true**, если восстановление выполнено (есть данные для восстановления);  - функция **push\_chain:** помещает реверс цепочки в стек автомата**,** всегда возвращает **true**;  - функция **step:** выполняет такт работы автомата, формирует диагностические сообщения, осуществляет отладочный вывод на консоль;  - функция **start:** запускает работу автомата, в цикле выполняет функцию **step**, осуществляет вывод диагностических сообщений;  - функция **savediagnosis:** сохраняет в массиве **diagnosis** строку диагностики; в массиве **diagnosis** сохраняются диагностические сообщения в порядке убывания позиции ленты (вызвавшей диагностику) и только в равным длине массива (макрос **MFST\_DIAGN\_NUMBER**). |
| MfstDiagnosis | Структура (внутренняя для **Mfst**): представление строки диагностики.  Структура включает:  - позиция входной ленты: **lenta\_position** (short);  - код возврата, сформированный функцией **step**;  - номер действующего на момент диагностики правила грамматики: **nrule** (short);  - номер текущей цепочки действующего на момент диагностики правила грамматики: **nrule\_chain**(short);  - два конструктора. |

**4.5 Описание алгоритма синтаксического разбора**

Синтаксический разбор выполняется по алгоритму, представленному ниже.

* Первым делом инициализируется магазин, в который помещается стартовый символ;
* Формируется входная лента по полученной ранее таблице лексем;
* Встретившийся нетерминальный символ раскрывается в соответствии с правилами, и реверс полученной цепочки записывается в магазин;
* Если терминалы на вершине стека и в ленте совпадают, то лента продвигается, а терминал удаляется с вершины стека. Иначе происходит возврат к предыдущему сохраненному состоянию и выбирается другое правило;
* Если в магазине встретился нетерминал, то осуществляется переход в пункт 3;
* Если достигнуто дно стека и при этом входная цепочка пуста, то значит синтаксический анализ выполнен успешно. Если нет, то 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**

В ситуации возникновения ошибки в синтаксическом анализаторе формируется ошибка в формате: код ошибки, текст сообщения, строка в исходном коде программы и позиция.

Все сообщения синтаксического анализатора с кодом ошибки представлены в таблице 4.3.

Таблица 4.3 – Сообщения синтаксического анализатора с кодом ошибки

|  |  |
| --- | --- |
| Номер ошибки | Текст сообщение |
| 600 | Неверная структура программы |
| 601 | Ошибка в теле функции |
| 602 | Ошибка в выражении |
| 603 | Ошибка в параметрах функции |
| 604 | Ошибка в параметрах вызываемой функции |
| 605 | Ошибка в подвыражении |
| 606 | Неверная структура условия |
| 607 | Ошибка в логическом подвыражении |

**4.8 Принцип обработки ошибок**

Синтаксический анализатор использует следующий алгоритм обработки ошибок:

* синтаксический анализатор пытается подобрать цепочку грамматики для текущей конструкции таблицы лексем;
* если подобрать цепочку невозможно, то генерируется соответствующая ошибка;
* все ошибки записываются в общую структуру ошибок;
* если была найдена ошибка после выполнения всего синтаксического анализа, то выводится соответствующее сообщение об ошибке, транслятор завершает свою работу.

**4.9 Контрольный пример**

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

**5 Разработка семантического анализатора**

**5.1 Структура семантического анализатора**

Семантический анализатор нужен для проверки смысловой структуры программы и ее семантической правильности, для обнаружения ошибок, которые связаны с типами данных, допустимыми операциями над переменными определенного типа данных, проверки совпадения возвращаемого значения функции с ее типом и т.д. Некоторые семантические проверки производятся на стадии лексического анализа. Входными параметрами семантического анализатора является таблица лексем и таблица идентификаторов. Выходным параметром является протокол работы.

Общая структура семантического анализатора представлена на рисунке 5.1.

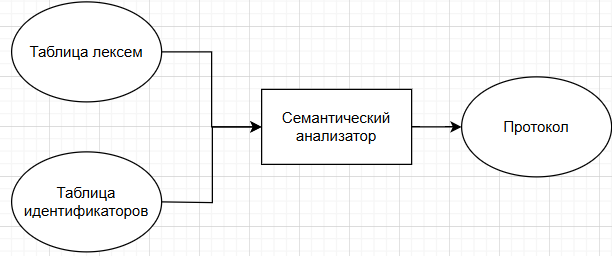


Рисунок 5.1 – Структура семантического анализатора

**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**

Перечень сообщений семантического анализатора с кодом ошибки представлен в таблице 5.1.

Таблица 5.1 – Перечень сообщений семантического анализатора

|  |  |
| --- | --- |
| Номер ошибки | Текст сообщение |
| 313 | Недопустимый размер целочисленного литерала |
| 314 | Типы данных в выражении не совпадают |
| 315 | Тип функции и возвращаемого значения не совпадают |
| 316 | Недопустимое строковое выражение справа от знака \'=\' |
| 317 | Неверное условное выражение |
| 318 | Недопустимо деление на ноль |

**5.4 Принцип обработки ошибок**

и обнаружении ошибки в исходном коде программы семантический анализатор формирует сообщение об ошибке и выводит его на консоль и в файл протокола .log. Транслятор завершает работу.

## **5.5 Контрольный пример**

Соответствие примеров некоторых ошибок в исходном коде и диагностических сообщений об ошибках приведено в таблице 5.2.

Таблица 5.2. – Примеры диагностики ошибок

|  |  |
| --- | --- |
| main{  integer x = 9;  string y = x;  } | Ошибка №314: Семантическая ошибка: Типы данных в выражении не совпадают Строка: 3 |
| main{  integer x = 9;  }  main{  string y = "qwerty";  } | Ошибка №302: Семантическая ошибка: Обнаружено несколько точек входа main Строка: 0 |

Ошибки записываются в протокол с указанием номера ошибки и диагностического сообщения. Анализ программы продолжается до обнаружения всех ошибок, после чего процесс анализа останавливается.

**6. Вычисление выражений**

**6.1 Выражения, допускаемые языком**

В языке LDV-2024 допускаются вычисления выражений целочисленного типа данных с поддержкой вызова функций внутри выражений. Приоритет операций представлен на таблице 6.1.

Таблица 6.1. – Приоритеты операций

|  |  |
| --- | --- |
| Операция | Значение приоритета |
| ( ) | 3 |
| \* | 2 |
| / | 2 |
| % | 2 |
| + | 1 |
| - | 1 |

Операторы в скобках имеют наивысший приоритет, за ними следуют умножение и деление, а затем сложение и вычитание.

**6.2 Польская запись и принцип её построения**

Польская запись является альтернативным способом записи арифметических выражений, которые затем удобно вычислять с помощью стека. Обратная польская запись – это форма записи выражений, когда операнды идут перед знаками операций.

Алгоритм построения обратной польской записи:

* если встретился операнд, то он переносится в строку результата преобразования;
* если встретился вызов функции, то он помещается в стек;
* если встретилась операция, то, если стек не пуст, сравнивается приоритет текущей операции с приоритетом операции на вершине стека. Пока приоритет операции, которая находится на вершине стека выше или равен приоритету текущей операции, из стека извлекается операция и помещается в результирующую строку. Текущая операция помещается в стек. Иначе, если изначально стек был пуст, то текущая операция сразу помещается в стек;
* если встретилась открывающаяся скобка, то она помещается в стек;
* если встретилась закрывающаяся скобка, то извлекаются все операции в результирующую строку, пока не встретится открывающаяся скобка. После этого открывающаяся скобка также извлекается из стека;
* если выражение рассмотрено, но стек не пуст, то извлекаются все символы из стека в результирующую строку.

**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**

Трансляция языка LDV-2024 производится в язык ассемблера. Структура генератора кода LDV-2024 представлена на рисунке 7.1.

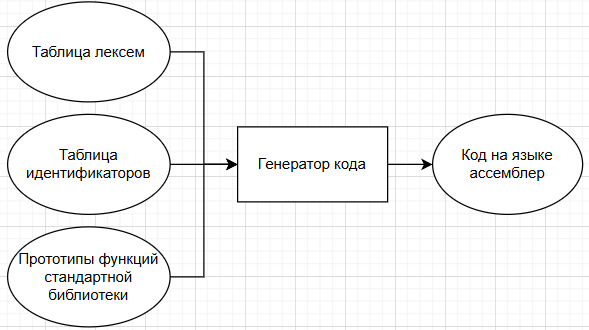


Рисунок 7.1 – Структура генератора кода

Генерация базируется на том, что каждый оператор представляет собой заранее заданный набор команд, который будет генерироваться при каждом использовании этого оператора. Функции на исходном языке программирования LDV-2024 транслируются в процедуры на ассемблере, возврат значений происходит через регистры и стек. При генерации условного оператора используются метки и переходы по ним.

**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 LDV-2024 и на языке ассемблера приведены в таблице 7.1.

Таблица 7.1 – Соответствия типов идентификаторов языка и ассемблера

|  |  |  |
| --- | --- | --- |
| Тип идентификатора на языке LDV-2024 | Тип идентификатора на языке ассемблера | Пояснение |
| integer | sdword | Хранит целочисленный тип данных. |
| string | byte | Каждый символ строки типа str хранится в поле размером 1 байт. |

Следовательно, таблица 7.1 показывает соответствия между типами идентификаторов на языке LDV-2024 и языке ассемблера. Это важно при переводе кода с языка LDV-2024 на язык ассемблера, чтобы правильно определить типы данных и использовать соответствующие инструкции и регистры для работы с идентификаторами.

**7.3 Статическая библиотека**

В языке LDV-2024 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 outw(char\* value)  int outwf(char\* value) | Вывод на консоль строки value  Вывод с переходом на новую строку |
| int outn(int value)  int outnf(int value) | Вывод на консоль целочисленной переменной value  Вывод с переходом на новую строку |
| int len(char\* source) | Вычисление длины строки |
| int comp(char\* source1, char\* source2) | сравнение строк source и str2 |

На этапе генерации кода в языке ассемблера библиотека подключается с использованием директивы includelib. Затем, с помощью оператора EXTRN, объявляются имена функций из этой библиотеки.

**7.4 Особенности алгоритма генерации кода**

В языке LDV-2024 генерация кода строится на основе таблиц лексем и идентификаторов. Общая схема работы генератора кода представлена на рисунке

Рисунок 7.2 – Структура генератора кода

**7.5 Параметры, управляющие генерацией кода**

На вход генератору кода поступают таблицы лексем и идентификаторов исходного код программы на языке LDV-2024. 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.

**8. Тестирование транслятора**

**8.1 Общие положения**

Тесты используются для проверки транслятора на работоспособность и для выявления недочетов и ошибок, а также последующего их исправления.

При возникновении ошибки транслятор завершает свою работу и выводит сообщение об ошибке с соответствующим ей кодом на консоль и в файл протокола .log.

**8.2 Результаты тестирования**

В языке LDV-2024 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  {  ё  integer а = 2;  …  } | Ошибка 111: Не доступный символ в исходном файле(-in) Строка: 13 Позиция в строке: 2 |

На этапе лексического анализа в языке LDV-2024 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  {…  integer NUM2 = 1;  …  } | Ошибка 201: Лексический анализатор: Неверное имя идентификатора строка 14 позиция 4 |

На этапе синтаксического анализа в языке LDV-2024 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 |
| main  {  integer x;}  } | Ошибка 601: Синтаксическая ошибка: Ошибка в теле функции.  601: строка 14, Синтаксическая ошибка: Ошибка в теле функции |
| string function fi({}  main{} | Ошибка 603: Синтаксическая ошибка: Ошибка в параметрах функции 603: строка 12, Синтаксическая ошибка: Ошибка в параметрах функции |
| string function fi()  {  output  } | Ошибка 602: Синтаксическая ошибка: Ошибка в выражении  602: строка 15, Синтаксическая ошибка: Ошибка в выражении |
| main  {  sum(4, 5;  } | Ошибка 600: Синтаксическая ошибка: Неверная структура программы 600: строка 12, Синтаксическая ошибка: Неверная структура программы |
| main {  integer x = 1;  integer y = 2;  if: x > y  } | Ошибка 606: Синтаксическая ошибка: Неверная структура условия  606: строка 19, Синтаксическая ошибка: Неверная структура условия |

Семантический анализ в языке LDV-2024 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 |
| string function fi(){} | Ошибка 301: Семантическая ошибка: Отсутствует точка входа main |
| main{}  main{} | Ошибка 302: Семантическая ошибка: Обнаружено несколько точек входа main |
| main  {  integer x = 1;  integer x = 2;  } | Ошибка 305: Семантическая ошибка: Попытка переопределения идентификатора cтрока 4 позиция 0 |

Продолжение таблицы 8.4

|  |  |
| --- | --- |
| Исходный код | Диагностическое сообщение |
| integer function fi(integer x){return x;}  main{fi();} | Ошибка 308: Семантическая ошибка: Кол-во ожидаемых и переданных функция и параметров не совпадают cтрока 3 позиция 0 |
| integer function fi(integer x){return x;}  main  {  string a = “1”;  fi(a);  } | Ошибка 309: Семантическая ошибка: Несовпадение типов передаваемых параметров cтрока 6 позиция 0 |
| main{  string x = "";  } | Ошибка 310: Семантическая ошибка: Использование пустого строкового литерала недопустимо cтрока 2 позиция 0 |
| main{  string x = ";  } | Ошибка 311: Семантическая ошибка: Не закрыт строковый литерал cтрока 2 позиция 0 |
| string function fi()  {  integer x = 5;  return x;  } | Ошибка 315: Семантическая ошибка: Тип функции и возвращаемого значения не совпадают cтрока 4 |
| main  {  string x;  x = "abc" + "d";  } | Ошибка 316: Семантическая ошибка: Недопустимое строковое выражение справа от знака '=' cтрока 4 |

# Заключение

В ходе выполнения курсовой работы был разработан транслятор и генератор кода для языка программирования LDV-2024 со всеми необходимыми компонентами. Таким образом, были выполнены основные задачи данной курсовой работы:

* Сформулирована спецификация языка LDV-2024;
* 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 LDV-2024 включает:

* 2 типа данных;
* Поддержка оператора вывода строки;
* Возможность вызова функций стандартной библиотеки;
* Наличие 5 арифметических операторов для вычисления выражений;
* Поддержка функций, процедур и оператора условия;
* Структурированная и классифицированная система для обработки ошибок пользователя.

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

2. Построение компиляторов / Никлаус Вирт 2010. – 194 с.

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

4. Язык программирования C++.Лекции и упражнения [6-е издание] / Стивен Прата 2019 – 1094 с.

**Приложение А**

Контрольный пример

integer function find\_max(integer x, integer y){

integer max;

if: x>y

{max = x;}

^{max = y;}

return max;

}

string function foo(string xxx){

return xxx;

}

void function func(){

println "Это процедура";

return;

}

main

{

func();

integer max = find\_max(2, 9);

string text = "Максимальное число больше 5";

if:max>5

{println text;}

string res = foo("sdfgh");

println res;

string str1 = "Project";

string str2 = "Projec!";

integer strc1 = len(str1);

integer strc2 = len(str2);

if: strc1 ~ strc2

{text = "Длины строк равны";}

^{text = "Длины строк НЕ равны";}

println text;

integer answer1 = comp(str1, str2);

integer answer2 = comp(str1, str1);

print answer1;

print " ";

println answer2;

integer math1 = 5 - 2;

integer math2 = 5 + 2;

integer math3 = 5 % 2;

integer math4 = 5 \* 2;

integer math5 = 5 / 2;

println math1;

println math2;

println math3;

println math4;

println math5;

integer check1 = 16xA;

integer check2 = 16x20;

integer res2 = check1 + check2;

println check1;

println check2;

println res2;

}

**Приложение Б**

Листинг 1 – Таблица идентификаторов контрольного примера

------------------Таблица индетификаторов------------------

Идентификатор | тип данных | первое вхождение | содержание

\_\_\_\_\_Литерал\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

L1 | string | 56 | [13]Это процедура

L2 | integer | 72 | 2

L3 | integer | 74 | 9

L4 | string | 80 | [27]Максимальное число больше 5

L5 | integer | 86 | 5

L6 | string | 98 | [5]sdfgh

L7 | string | 107 | [7]Project

L8 | string | 112 | [7]Projec!

L9 | string | 139 | [17]Длины строк равны

L10 | string | 146 | [20]Длины строк НЕ равны

L11 | string | 176 | [1]

L12 | integer | 231 | 1

L13 | integer | 236 | 32

\_\_\_\_\_Переменная\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

find\_maxmax | integer | 12 | -

max | integer | 68 | -

text | string | 78 | -

res | string | 94 | -

str1 | string | 105 | -

str2 | string | 110 | -

strc1 | integer | 115 | -

strc2 | integer | 123 | -

answer1 | integer | 153 | -

answer2 | integer | 163 | -

math1 | integer | 182 | -

math2 | integer | 189 | -

math3 | integer | 196 | -

math4 | integer | 203 | -

math5 | integer | 210 | -

check1 | integer | 229 | -

check2 | integer | 234 | -

res2 | integer | 239 | -

\_\_\_\_\_Параметр\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

find\_maxx | integer | 5 | -

find\_maxy | integer | 8 | -

fooxxx | string | 42 | -

source | string | -1 | -

source1 | string | -1 | -

source2 | string | -1 | -

\_\_\_\_\_Функция\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

find\_max | integer | 2 | -

foo | string | 39 | -

func | void | 51 | -

len | integer | 117 | -

comp | integer | 155 | -

\_\_\_\_\_Оператор\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

- | - | 185 | -

+ | - | 192 | -

% | - | 199 | -

\* | - | 206 | -

/ | - | 213 | -

+ | - | 242 | -

\_\_\_\_\_Логический оператор\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

> | - | 17 | -

> | - | 85 | -

~ | - | 133 | -

**Приложение В**

Листинг 1 – Грамматика языка LDV-2024

Greibach greibach(

NS('S'),

TS('$'),

11,

Rule(

NS('S'),

GRB\_ERROR\_SERIES + 0,

6,

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

Rule::Chain(7, TS('t'), TS('f'), TS('i'), TS('('), TS(')'), NS('T'), NS('S')),

Rule::Chain(8, TS('t'), TS('f'), TS('i'), TS('('), NS('F'), TS(')'), NS('T'), NS('S')),

Rule::Chain(7, TS('n'), TS('f'), TS('i'), TS('('), TS(')'), NS('T'), NS('S')),

Rule::Chain(8, TS('n'), TS('f'), TS('i'), TS('('), NS('F'), TS(')'), NS('T'), NS('S')),

Rule::Chain(5, TS('m'), TS('{'), NS('N'), TS('}'), NS('S'))

),

Rule(

NS('T'),

GRB\_ERROR\_SERIES + 0,

4,

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

Rule::Chain(5, TS('{'), NS('N'), TS('r'), TS(';'), TS('}')),

Rule::Chain(6, TS('{'), NS('N'), TS('r'), NS('E'), TS(';'), TS('}')),

Rule::Chain(5, TS('{'), TS('r'), NS('E'), TS(';'), TS('}'))

),

Rule(

NS('N'),

GRB\_ERROR\_SERIES + 1,

24,

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

Rule::Chain(6, TS('i'), TS('('), NS('W'), TS(')'), TS(';'), NS('N')),

Rule::Chain(5, TS('i'), TS('('), NS('W'), TS(')'), TS(';')),

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

Rule::Chain(4, TS('i'), TS('('), TS(')'), TS(';')),

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

Rule::Chain(5, TS('t'), TS('i'), TS('='), NS('E'), TS(';')),

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

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

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

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

Rule::Chain(4, TS('p'), TS('l'), TS(';'), NS('N')),

Rule::Chain(4, TS('z'), TS('l'), TS(';'), NS('N')),

Rule::Chain(4, TS('p'), NS('E'), TS(';'), NS('N')),

Rule::Chain(4, TS('z'), NS('E'), TS(';'), NS('N')),

Rule::Chain(3, TS('p'), NS('E'), TS(';')),

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

Rule::Chain(3, TS('t'), TS('i'), TS(';')),

Rule::Chain(3, TS('p'), TS('i'), TS(';')),

Rule::Chain(3, TS('z'), TS('i'), TS(';')),

Rule::Chain(3, TS('p'), TS('l'), TS(';')),

Rule::Chain(3, TS('z'), TS('l'), TS(';')),

Rule::Chain(4, TS('w'), NS('I'), TS('|'), NS('X')),

Rule::Chain(5, TS('w'), NS('I'), TS('|'), NS('X'), NS('N'))

),

Rule(

NS('B'),

GRB\_ERROR\_SERIES + 6,

11,

Rule::Chain(5, TS('t'), TS('i'), TS('='), NS('E'), TS(';')),

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

Rule::Chain(3, TS('p'), NS('E'), TS(';')),

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

Rule::Chain(3, TS('t'), TS('i'), TS(';')),

Rule::Chain(3, TS('p'), TS('i'), TS(';')),

Rule::Chain(3, TS('z'), TS('i'), TS(';')),

Rule::Chain(3, TS('p'), TS('l'), TS(';')),

Rule::Chain(3, TS('z'), TS('l'), TS(';')),

Rule::Chain(4, TS('w'), NS('I'), TS('|'), NS('X')),

Rule::Chain(5, TS('w'), NS('I'), TS('|'), NS('X'), NS('N'))

),

Rule(

NS('X'),

GRB\_ERROR\_SERIES + 6,

2,

Rule::Chain(7, TS('{'), NS('N'), TS('}'), TS('^'), TS('{'), NS('N'), TS('}')),

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

),

Rule(

NS('O'),

GRB\_ERROR\_SERIES + 7,

1,

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

),

Rule(

NS('E'),

GRB\_ERROR\_SERIES + 2,

10,

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

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

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

Rule::Chain(4, TS('i'), TS('('), NS('W'), TS(')')),

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

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(5, TS('i'), TS('('), NS('W'), TS(')'), NS('M')),

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

),

Rule(

NS('F'),

GRB\_ERROR\_SERIES + 3,

2,

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

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

),

Rule(

NS('W'),

GRB\_ERROR\_SERIES + 4,

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 + 2,

2,

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

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

),

Rule(

NS('I'),

GRB\_ERROR\_SERIES + 6,

2,

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

Rule::Chain(6, TS(':'), NS('E'), TS('q'), NS('E'), TS('&'), NS('I'))

)

);

Листинг 2 – Структура магазинного автомата

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 prc\_step, short pnrule, short pnrule\_chain);

}diagnosis[MFST\_DIAGN\_NUMBER];

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

public:

using std::stack<MfstState>::c;

};

GRBALPHABET\* lenta;

short lenta\_position;

short nrule;

short nrulechain;

short lenta\_size;

GRB::Greibach greibach;

LT::LexTable lex;

bool more = false;

Log::LOG log;

MFSTSTSTACK st;

my\_stack\_MfstState storestate;

Mfst();

Mfst(LT::LexTable& plex, GRB::Greibach pgreibach);

char\* getCSt(char\* buf);

char\* getCLenta(char\* buf, short pos, short n = 25);

char\* getDiagnosis(short n, char\* buf);

bool savestate();

bool resetstate();

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

RC\_STEP step();

bool start();

bool saveddiagnosis(RC\_STEP pprc\_step);

void printrules();

struct Deducation {

short size;

short\* nrules;

short\* nrulechains;

Deducation() {

size = 0;

nrules = 0;

nrulechains = 0;

}

}deducation;

bool savededucation();

};

Листинг 3 – Структура грамматики Грейбах

struct Greibach //грамматика Грейбах

{

short size; //количество правил

GRBALPHABET startN; //стартовый символ

GRBALPHABET stbottomT;//дно стека

Rule\* rules; //множество правил

Greibach() { short size = 0; startN = 0; stbottomT = 0; rules = 0; };

Greibach(

GRBALPHABET pstartN, //стартовый символ

GRBALPHABET pstbottom, //дно стека

short psize, //количество правил

Rule r, ... //правила

);

short getRule( //получить правило, возвращается номер правила или -1

GRBALPHABET pnn, //левый символ правила

Rule& prule //возвращаемое правило грамматики

);

Rule getRule(short n); //получить правило по номеру

};

Greibach getGreibach(); //получить грамматику

Листинг 4 – Разбор исходного кода синтаксическим анализатором

Шаг :Правило Входная лента Стек

0 : S->tfi()TS tfi(ti,ti){ti;w:iqi|{i=i; S$

0 : SAVESTATE: 1

0 : tfi(ti,ti){ti;w:iqi|{i=i; tfi()TS$

1 : fi(ti,ti){ti;w:iqi|{i=i;} fi()TS$

2 : i(ti,ti){ti;w:iqi|{i=i;}^ i()TS$

3 : (ti,ti){ti;w:iqi|{i=i;}^{ ()TS$

4 : ti,ti){ti;w:iqi|{i=i;}^{i )TS$

5 : TS\_NOK / NS\_NORULECHAIN

5 : RESSTATE

5 : tfi(ti,ti){ti;w:iqi|{i=i; S$

6 : S->tfi(F)TS tfi(ti,ti){ti;w:iqi|{i=i; S$

6 : SAVESTATE: 1

6 : tfi(ti,ti){ti;w:iqi|{i=i; tfi(F)TS$

7 : fi(ti,ti){ti;w:iqi|{i=i;} fi(F)TS$

8 : i(ti,ti){ti;w:iqi|{i=i;}^ i(F)TS$

9 : (ti,ti){ti;w:iqi|{i=i;}^{ (F)TS$

10 : ti,ti){ti;w:iqi|{i=i;}^{i F)TS$

11 : F->ti ti,ti){ti;w:iqi|{i=i;}^{i F)TS$

11 : SAVESTATE: 2

11 : ti,ti){ti;w:iqi|{i=i;}^{i ti)TS$

12 : i,ti){ti;w:iqi|{i=i;}^{i= i)TS$

13 : ,ti){ti;w:iqi|{i=i;}^{i=i )TS$

Листинг 4 (прод.) – Разбор исходного кода синтаксическим анализатором

242: E->i(W)M i;} E;N}$

2242: SAVESTATE: 109

2242: i;} i(W)M;N}$

2243: ;} (W)M;N}$

2244: TS\_NOK / NS\_NORULECHAIN

2244: RESSTATE

2244: i;} E;N}$

2245: E->i()M i;} E;N}$

2245: SAVESTATE: 109

2245: i;} i()M;N}$

2246: ;} ()M;N}$

2247: TS\_NOK / NS\_NORULECHAIN

2247: RESSTATE

2247: i;} E;N}$

2248: NS\_NRCHAIN/NS\_NR

2248: RESSTATE

2248: zi;} N}$

2249: N->zE; zi;} N}$

2249: SAVESTATE: 108

2249: zi;} zE;}$

2250: i;} E;}$

2251: E->i i;} E;}$

2251: SAVESTATE: 109

2251: i;} i;}$

2252: ;} ;}$

2253: } }$

2254: $

2255: 6

2256: ------>LENTA\_END

**Приложение Г**

Листинг 1 – Программная реализация механизма преобразования в ПОЛИЗ

bool PolishNotation(int i, Lex::LEX& lex)

{

std::stack<LT::Entry> stack; // В стек будем заносить операции

std::queue<LT::Entry> queue; // В очередь будем заносить операнды

LT::Entry placeholder\_symbol;

placeholder\_symbol.idxTI = -1;

placeholder\_symbol.lexema = ' ';

placeholder\_symbol.sn = lex.lextable.table[i].sn;

LT::Entry function\_symbol; // Лексема, обозначающая конец функции

function\_symbol.idxTI = LT\_TI\_NULLIDX; // Признак начала и окончания функции

function\_symbol.lexema = '@';

function\_symbol.sn = lex.lextable.table[i].sn;

int idx;

int lexem\_counter = 0;

int parm\_counter = 0; // Количество параметров в функции

int lexem\_position = i; // Запоминаем номер лексемы перед преобразованием

char\* buf = new char[i];

bool findFunc = false; // флаг на нахождение функции

for (i; lex.lextable.table[i].lexema != LEX\_SEMICOLON; i++, lexem\_counter++)

{

switch (lex.lextable.table[i].lexema)

{

case LEX\_ID:

case LEX\_LITERAL:

if (lex.idtable.table[lex.lextable.table[i].idxTI].idtype == IT::F)

{

findFunc = true;

idx = lex.lextable.table[i].idxTI;

}

else

{

if (findFunc)

parm\_counter++;

queue.push(lex.lextable.table[i]);

}

continue;

case LEX\_LEFTTHESIS:

stack.push(lex.lextable.table[i]);

continue;

case LEX\_RIGHTTHESIS:

while (stack.top().lexema != LEX\_LEFTTHESIS)

{

queue.push(stack.top());

stack.pop();

if (stack.empty())

return false;

}

if (!findFunc)

stack.pop();

else {

function\_symbol.idxTI = idx;

idx = LT\_TI\_NULLIDX;

lex.lextable.table[i] = function\_symbol;

queue.push(lex.lextable.table[i]);

\_itoa\_s(parm\_counter, buf, 2, 10);

stack.top().lexema = buf[0];

stack.top().idxTI = LT\_TI\_NULLIDX;

stack.top().sn = function\_symbol.sn;

queue.push(stack.top());

stack.pop();

parm\_counter = 0;

findFunc = false;

}

continue;

case LEX\_OPERATOR:

while (!stack.empty() && lex.lextable.table[i].priority <= stack.top().priority)

{

queue.push(stack.top());

stack.pop();

}

stack.push(lex.lextable.table[i]);

continue;

}

}

while (!stack.empty())

{

if (stack.top().lexema == LEX\_LEFTTHESIS || stack.top().lexema == LEX\_RIGHTTHESIS)

return false;

queue.push(stack.top());

stack.pop();

}

while (lexem\_counter != 0)

{

if (!queue.empty())

{

lex.lextable.table[lexem\_position++] = queue.front();

queue.pop();

}

else

lex.lextable.table[lexem\_position++] = placeholder\_symbol;

lexem\_counter--;

}

for (int i = 0; i < lexem\_position; i++)

{

if (lex.lextable.table[i].lexema == LEX\_OPERATOR || lex.lextable.table[i].lexema == LEX\_LITERAL)

lex.idtable.table[lex.lextable.table[i].idxTI].idxfirstLE = i;

}

return true;

}

**Приложение Д**

Листинг 1 – Результат генерации кода контрольного примера в Ассемблере

|  |  |
| --- | --- |
| .586  .model flat, stdcall  includelib libucrt.lib  includelib kernel32.lib  includelib "../Debug/lib.lib"  ExitProcess PROTO:DWORD  .stack 4096  ExitProcess PROTO :DWORD  outn PROTO : SDWORD  outw PROTO : DWORD  outnf PROTO : SDWORD  outwf PROTO : DWORD  len PROTO : DWORD  comp PROTO : DWORD, : DWORD  stcmp PROTO : DWORD, : DWORD  .const  newline byte 13, 10, 0  L1 byte 'Это процедура', 0  L2 sdword 2  L3 sdword 9  L4 byte 'Максимальное число больше 5', 0  L5 sdword 5  L6 byte 'sdfgh', 0  L7 byte 'Project', 0  L8 byte 'Projec!', 0  L9 byte 'Длины строк равны', 0  L10 byte 'Длины строк НЕ равны', 0  L11 byte ' ', 0  L12 sdword 1  L13 sdword 32  .data  temp sdword ?  buffer byte 256 dup(0)  find\_maxmax sdword 0  max sdword 0  text dword ?  res dword ?  str1 dword ?  str2 dword ?  strc1 sdword 0  strc2 sdword 0  answer1 sdword 0  answer2 sdword 0  int\_to\_char ENDP  ;----------- find\_max ------------  find\_max PROC,  find\_maxx : sdword, find\_maxy : sdword  ; --- save registers ---  push ebx  push edx  ; ----------------------  push find\_maxx  pop ebx  mov left, ebx  push find\_maxy  pop ebx  mov rig, ebx  mov edx, left  cmp edx, rig  jg right1  jl wrong1  right1:  push find\_maxx  pop ebx  mov find\_maxmax, ebx  jmp next1  wrong1:  push find\_maxy  pop ebx  mov find\_maxmax, ebx  next1:  ; --- restore registers ---  pop edx  pop ebx  ; -------------------------  mov eax, find\_maxmax  ret  find\_max ENDP  ;------------------------------  ;----------- foo ------------  foo PROC,  fooxxx : dword  ; --- save registers ---  push ebx  pop ebx  mov rig, ebx  mov edx, left  cmp edx, rig  jg right2  jmp next2  right2:  INVOKE outwf, text  next2:  mov res, offset L6  INVOKE outwf, res  mov str1, offset L7  mov str2, offset L8  push str1  call len  push eax  pop ebx  mov strc1, ebx  push str2  call len  push eax  pop ebx  mov strc2, ebx  push strc1  pop ebx  mov left, ebx  push strc2  pop ebx  mov rig, ebx  mov edx, left  cmp edx, rig  jz right3  jnz wrong3  right3:  mov text, offset L9  jmp next3  wrong3:  mov math2, ebx  push L5  push L2  pop ebx  pop eax  test ebx, ebx  jz EXIT\_DIV\_ON\_NULL  cdq  mov edx, 0  idiv ebx  push edx  pop ebx  mov math3, ebx  push L5  push L2  pop ebx  pop eax  imul eax, ebx  push eax  pop ebx  mov math4, ebx  push L5  push L2  pop ebx  pop eax  test ebx, ebx  jz EXIT\_DIV\_ON\_NULL  cdq  mov edx, 0  idiv ebx  push eax  pop ebx  mov math5, ebx  mov eax, math2  mov result, eax  INVOKE int\_to\_char, offset result\_str, result  INVOKE outwf, offset result\_str  mov eax, math3  mov result, eax  INVOKE int\_to\_char, offset result\_str, result  INVOKE outwf, offset result\_str  mov eax, math4 | math1 sdword 0  math2 sdword 0  math3 sdword 0  math4 sdword 0  math5 sdword 0  check1 sdword 0  check2 sdword 0  res2 sdword 0  left dword ?  rig dword ?  result sdword ?  result\_str byte 4 dup(0)  .code  int\_to\_char PROC uses eax ebx ecx edi esi,  pstr: dword,  intfield : sdword  mov edi, pstr  mov esi, 0  mov eax, intfield  cdq  mov ebx, 10  idiv ebx  test eax, 80000000h  jz plus  neg eax  neg edx  mov cl, '-'  mov[edi], cl  inc edi  plus :  push dx  inc esi  test eax, eax  jz fin  cdq  idiv ebx  jmp plus  fin :  mov ecx, esi  write :  pop bx  add bl, '0'  mov[edi], bl  inc edi  loop write  ret  push edx  ; ----------------------  ; --- restore registers ---  pop edx  pop ebx  ; -------------------------  mov eax, fooxxx  ret  foo ENDP  ;------------------------------  ;----------- func ------------  func PROC    ; --- save registers ---  push ebx  push edx  ; ----------------------  INVOKE outwf, offset L1  ; --- restore registers ---  pop edx  pop ebx  ; -------------------------  ret  func ENDP  ;------------------------------  ;----------- MAIN ------------  main PROC  call func  push L2  push L3  call find\_max  push eax  pop ebx  mov max, ebx  mov text, offset L4  push max  pop ebx  mov left, ebx  push L5  mov text, offset L10  next3:  INVOKE outwf, text  push str1  push str2  call comp  push eax  pop ebx  mov answer1, ebx  push str1  push str1  call comp  push eax  pop ebx  mov answer2, ebx  mov eax, answer1  mov result, eax  INVOKE int\_to\_char, offset result\_str, result  INVOKE outw, offset result\_str  INVOKE outw, offset L11  mov eax, answer2  mov result, eax  INVOKE int\_to\_char, offset result\_str, result  INVOKE outwf, offset result\_str  push L5  push L2  pop ebx  pop eax  sub eax, ebx  push eax  pop ebx  mov math1, ebx  push L5  push L2  pop ebx  pop eax  add eax, ebx  push eax  pop ebx  mov result, eax  INVOKE int\_to\_char, offset result\_str, result  INVOKE outwf, offset result\_str  mov eax, math5  mov result, eax  INVOKE int\_to\_char, offset result\_str, result  INVOKE outwf, offset result\_str  push L12  pop ebx  mov check1, ebx  push L13  pop ebx  mov check2, ebx  push check1  push check2  pop ebx  pop eax  add eax, ebx  push eax  pop ebx  mov res2, ebx  mov eax, check1  mov result, eax  INVOKE int\_to\_char, offset result\_str, result  INVOKE outwf, offset result\_str  mov eax, check2  mov result, eax  INVOKE int\_to\_char, offset result\_str, result  INVOKE outwf, offset result\_str  mov eax, res2  mov result, eax  INVOKE int\_to\_char, offset result\_str, result  INVOKE outwf, offset result\_str  INVOKE ExitProcess,0  EXIT\_DIV\_ON\_NULL:  INVOKE ExitProcess,-1  main ENDP  end main |