МИНИСТЕРСТВО ОБРАЗОВАНИЯ РЕСПУБЛИКИ БЕЛАРУСЬ

Учреждение образования «БЕЛОРУССКИЙ ГОСУДАРСТВЕННЫЙ

ТЕХНОЛОГИЧЕСКИЙ УНИВЕРСИТЕТ»

Факультет Информационных Технологий

Кафедра Программной инженерии

Специальность 1-40 01 01 Программное обеспечение информационных технологий

Специализация Программирование интернет-приложений

**ПОЯСНИТЕЛЬНАЯ ЗАПИСКА**

**К КУРСОВОМУ ПРОЕКТУ НА ТЕМУ:**

«Разработка компилятора LAD-2022»

Выполнил студент Ломако Александр Дмитриевич

(Ф.И.О.)

Руководитель проекта преп.-стажер Карпович М. Н.

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

Заведующий кафедрой к.т.н., доц. Пацей Н.В.

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

Консультанты преп.-стажер Карпович М. Н.

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

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

Нормоконтролер преп.-стажер Карпович М. Н.

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

Курсовой проект защищен с оценкой

Минск 2022

**Содержание**

[Введение 4](#_Toc122471144)

[1. Спецификации языка программирования 5](#_Toc122471145)

[1.1 Характеристика языка программирования 5](#_Toc122471146)

[1.2 Алфавит языка 5](#_Toc122471147)

[1.3 Символы сепараторы 5](#_Toc122471148)

[1.4 Применяемые кодировки 6](#_Toc122471149)

[1.5 Типы данных 7](#_Toc122471150)

[1.7 Преобразование типов данных 7](#_Toc122471151)

[1.8 Идентификаторы 7](#_Toc122471152)

[1.9 Литералы 8](#_Toc122471153)

[1.10 Объявление данных 8](#_Toc122471154)

[1.11 Инициализация данных 9](#_Toc122471155)

[1.12 Инструкции языка 9](#_Toc122471156)

[1.13 Операции языка 10](#_Toc122471157)

[1.14 Выражения и их вычисления 10](#_Toc122471158)

[1.15 Программные конструкции языка 11](#_Toc122471159)

[1.16 Область видимости 12](#_Toc122471160)

[1.17 Семантические проверки 12](#_Toc122471161)

[1.18 Распределение оперативной памяти на этапе выполнения 12](#_Toc122471162)

[1.19 Стандартная библиотека и её состав 13](#_Toc122471163)

[1.20 Ввод и вывод данных 14](#_Toc122471164)

[1.21 Точка входа 14](#_Toc122471165)

[1.22 Препроцессор 14](#_Toc122471166)

[1.23 Соглашения о вызовах 14](#_Toc122471167)

[1.24 Объектный код 14](#_Toc122471168)

[1.25 Классификация сообщений транслятора 14](#_Toc122471169)

[1.26 Контрольный пример 15](#_Toc122471170)

[2. Структура транслятора 16](#_Toc122471171)

[2.1 Компоненты транслятора, их назначение и принципы взаимодействия 16](#_Toc122471172)

[2.2 Перечень входных параметров транслятора 17](#_Toc122471173)

[2.3 Перечень протоколов, формируемых транслятором и их содержимое 17](#_Toc122471174)

[3. Разработка лексического анализатора 18](#_Toc122471175)

[3.1 Структура лексического анализатора 18](#_Toc122471176)

[3.2 Контроль входных символов 19](#_Toc122471177)

[3.3 Удаление избыточных символов 19](#_Toc122471178)

[3.4 Перечень ключевых слов 19](#_Toc122471179)

[3.5 Основные структуры данных 20](#_Toc122471180)

[3.6 Структура и перечень сообщений лексического анализатора 21](#_Toc122471181)

[3.7 Принцип обработки ошибок 21](#_Toc122471182)

[3.8 Параметры лексического анализатора 21](#_Toc122471183)

[3.9 Алгоритм лексического анализа 21](#_Toc122471184)

[3.10 Контрольный пример 22](#_Toc122471185)

[4. Разработка синтаксического анализатора 23](#_Toc122471186)

[4.1 Структура синтаксического анализатора 23](#_Toc122471187)

[4.2 Контекстно-свободная грамматика, описывающая синтаксис 23](#_Toc122471188)

[4.3 Построение конечного магазинного автомата 26](#_Toc122471189)

[4.4 Основные структуры данных 26](#_Toc122471190)

[4.5 Описание алгоритма синтаксического разбора 27](#_Toc122471191)

[4.6 Структура и перечень сообщений синтаксического анализатора 27](#_Toc122471192)

[4.7 Параметры синтаксического анализатора и режимы его работы 28](#_Toc122471193)

[4.9 Контрольный пример 28](#_Toc122471194)

[5. Разработка семантического анализатора 29](#_Toc122471195)

[5.1 Структура семантического анализатора 29](#_Toc122471196)

[5.2 Функции семантического анализатора 29](#_Toc122471197)

[5.3 Перечень сообщений семантического анализатора 29](#_Toc122471198)

[5.4 Принцип обработки ошибок 30](#_Toc122471199)

[5.5 Контрольный пример 30](#_Toc122471200)

[6. Преобразование выражений 31](#_Toc122471201)

[6.1 Выражения, допускаемые языком 31](#_Toc122471202)

[6.2 Польская запись 31](#_Toc122471203)

[6.3 Программная реализация обработки выражений 32](#_Toc122471204)

[6.4 Контрольный пример 32](#_Toc122471205)

[7. Генерация кода 33](#_Toc122471206)

[7.1 Структура генератора кода 33](#_Toc122471207)

[7.2 Представление типов данных в оперативной памяти 33](#_Toc122471208)

[7.3 Особенности алгоритма генерации кода 34](#_Toc122471209)

[7.4 Контрольный пример 34](#_Toc122471210)

[8. Тестирование транслятора 35](#_Toc122471211)

[8.1 Проверка на допустимость символов 35](#_Toc122471212)

[8.2 Тестирование лексического анализатора 35](#_Toc122471213)

[8.3 Тестирование синтаксического анализатора 36](#_Toc122471214)

[8.4 Тестирование семантического анализатора 36](#_Toc122471215)

[Заключение 38](#_Toc122471216)

[Список использованных источников 39](#_Toc122471217)

[Приложения 40](#_Toc122471218)

[Контрольный пример 40](#_Toc122471219)

[Приложение А 41](#_Toc122471220)

[Приложение Б 52](#_Toc122471221)

[Приложение В 58](#_Toc122471222)

[Приложение Г 62](#_Toc122471223)

[Приложение Е 67](#_Toc122471224)

# Введение

Задачей данного курсового проекта является разработка транслятора для своего языка программирования: LAD-2022.

Главная задача транслятора заключается в том, чтобы сделать программу, написанную на языке программирования LAD-2022, понятной компьютеру. В данном курсовом проекте исходный код транслируется на язык ассемблер. Язык ассемблер – тип языка программирования низкого уровня, представляющий собой формат записи машинных команд, удобный для восприятия человеком.

Исходя из цели курсового проекта, были определены следующие задачи:

* разработка спецификации языка программирования;
* разработка структуры транслятора;
* разработка лексического анализатора;
* разработка синтаксического анализатора;
* разработка семантического анализатора;
* обработка выражений;
* генерация кода на язык Assembler;
* тестирование транслятора.

Решения каждой из поставленных задач будут приведены в соответствующих главах курсового проекта.

# 1. Спецификации языка программирования

# 1.1 Характеристика языка программирования

Язык программирования LAD-2022 – это универсальный язык высокого уровня. Он является транслируемым, не объектно-ориентированным языком программирования. Данный язык строго типизируемый, что говорит о невозможности преобразования типов, транслируемым языком программирования.

# **1.2 Алфавит языка**

Алфавит языка программирования – совокупность символов, используемых в языке.

Алфавит языка LAD-2022 формально представлен как совокупность правил в форме Бэкуса-Наура в таблице 1.1.

Таблица 1.1 – Алфавит языка программирования LAD\_2022

|  |
| --- |
| Правило |
| <строчная буква латинского алфавита>::= a|b|c|d|e|f|g|h|i|j|k|l|m|n|o|p|q|r|s|t|u|v|w|x|y|z |
| <прописная буква латинского алфавита>::= A|B|C|D|E|F|G|H|I|J|K|L|M|N|O|P|Q|R|S|T|U|V|W|X|Y|Z |
| <цифра>::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
| <арифметическая операция>::= +|-|\*|/|% |
| <символ- сепаратор>::=,|(|)|;|{|}|?|$|’пробел’ |
| <символы сравнения>::=>|<|&|!|@|# |

Из данной таблицы мы можем заметить, что исходный код LAD-2022 может содержать символы латинского алфавита, цифры десятичной системы счисления от 0 до 9, символы арифметических операций, и символы сравнения, а также символы-разделители.

# 1.3 Символы сепараторы

Сепараторы необходимы для разделения операций языка. Сепараторы, используемые в языке программирования LAD-2022, приведены в таблице 1.2.

Таблица 1.2 – Сепараторы

|  |  |
| --- | --- |
| Сепаратор | Назначение |
| ‘пробел’ | Разделитель цепочек. Допускается везде кроме названий идентификаторов и ключевых слов |
| **{** … **}** | Блок функции или конструкции условия/цикла |
| **(** … **)** | Блок фактических или формальных параметров функции, а также приоритет арифметических операций |
| **$ … $** | Блок логического условия |

Продолжение таблицы 1.2

|  |  |
| --- | --- |
| Сепаратор | Назначение |
| **,** | Разделитель параметров функций |
| **;** | Разделитель программных конструкций |
| **?** | Символ, отделяющий условные конструкции |
| **=** | Оператор присваивания |
| **+ - \* /** % | Арифметические операции (сложение, вычитание, умножение, деление, остаток от деления) |
| **> < & ! @ #** | Логические операции (операции сравнения: больше, меньше, проверка на равенство, на неравенство), используемые в условии цикла/условной конструкции. |

Данная таблица показывает для какой операции предназначен тот или иной сепаратор в языке LAD-2022.

# 1.4 Применяемые кодировки

Для написания программ язык LAD-2022 использует кодировку ASCII, содержащую английский алфавит, а также некоторые специальные символы, такие как: ( ) , ; ? $ + - / % \* > < & ! @ # { }.



Рисунок 1.1 – кодировка ASCII

На данном рисунке представлена таблица кодировки ASCII, в языке LAD-2022 разрешены только латинские символы, следовательно разрешена только первая часть таблицы (первые 128 символов).

# 1.5 Типы данных

В LAD-2022 используется 4 типа данных: целочисленные (int), целочисленный беззнаковые (uint), строковые (str) и символьные (symbol). Пользовательские типы данных не поддерживаются. Допускается использование фундаментальных типов данных, определенных в таблице 1.3.

Таблица 1.3 – Типы данных языка LAD-2022

|  |  |  |  |
| --- | --- | --- | --- |
| Тип данных | Описание типа данных | Размер в байтах | Диапазон допустимых значений |
| int | Фундаментальный тип данных. Предусмотрен для объявления целочисленных данных. Автоматически инициализируется нулевым значением. | 2 байта | От –32768 до 32767 |
| uint | Фундаментальный тип данных. Предусмотрен для объявления целочисленных данных. Автоматически инициализируется нулевым значением. | 2 байта | От 0  до 65535 |
| str | Фундаментальный тип данных. Предусмотрен для объявления строк. Автоматическая инициализация строкой нулевой длины. | Последовательность в 255 символов, обязательно оканчивающаяся нуль-символом '\0' | - |
| symbol | Фундаментальный тип данных. Предусмотрен для объявления символов. Автоматически инициализируется символом ‘\0’. | 1 байт | - |

В данной таблице представлена информация о типах данных в языке LAD-2022. Легко заметить, что данный язык имеет 4 типа данных.

# 1.7 Преобразование типов данных

В языке программирования LAD-2022 преобразование типов данных не поддерживается.

# 1.8 Идентификаторы

В имени идентификатора допускаются символы латинского алфавита нижнего регистра. Перед именем каждого идентификаторы функции должно стоять ключевое слово «function».

Правило для построения идентификатора в форме Бэкуса-Наура представлено формулой **(1)**:

<идентификатор> ::= <значение> | <строчная буква латинского алфавита>{

(<строчная буква латинского алфавита>| <цифра > )} **(1)**

# 1.9 Литералы

В языке существует 3 типа литералов: целого, символьного и строкового типов.

Таблица 1.4 – Описание литералов

|  |  |
| --- | --- |
| Тип литерала | Описание |
| Литералы целого типа | Целочисленные неотрицательные литералы, инициализируются 0. Литералы только rvalue. |
| Символьные литералы | Символы, заключенные в ' '(одинарные кавычки), инициализируются нуль символом, символьные переменные. Только rvalue. |
| Строковые литералы | Символы, заключённые в '' ''(двойные кавычки), инициализируются пустой строкой, строковые переменные. Только rvalue. |

В таблице 1.4 представлено краткое описание литералов, которые представлены в языке программирования LAD-2022.

# 1.10 Объявление данных

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

Пример объявления числового типа с инициализацией:

new int a = -1;

new uint b = 563;

Пример объявления числового типа без инициализации:

new int a;

Пример объявления переменной строкового типа с инициализацией:

new str string = “hello world”;

Для объявления функций используется ключевое слово function, перед которым, в обязательном порядке, указывается тип функции, после задается имя функции. Далее обязателен список параметров и тело функции.

Примеры объявления функций:

int function funcint (int x, int y) { … }

str function funcstr (str a, str b) { … }

# 1.11 Инициализация данных

В момент объявления переменной она инициализируется фиксированным значением. Способы инициализации и их описание представлено в таблице 1.5.

Таблица 1.5 – Способы инициализации переменных

|  |  |
| --- | --- |
| Вид инициализации | Примечание |
| new <тип данных>  <идентификатор>; | Автоматическая инициализация: переменные типа int и uint инициализируются нулём, переменные типа str – пустой строкой, symbol – символом ‘\0’. |
| new <тип данных>  <идентификатор> =  <значение>; | Переменные инициализируются заданным значением, если оно такого же типа, что и переменная. |

В данной таблице показаны способы инициализации переменных, с автоматической инициализацией и с инициализацией значением.

# 1.12 Инструкции языка

Возможные инструкции языка программирования LAD-2022 представлены в общем виде в таблице 1.6.

Таблица 1.6 – Инструкции языка программирования LAD-2022

|  |  |
| --- | --- |
| Инструкция | Запись на языке LAD-2022 |
| Объявление  переменной | new <тип данных> <идентификатор>; |
| Объявление переменной с явной инициализацией | new <тип данных> <идентификатор> = <значение>;  Значение – инициализатор конкретного типа. Может быть только литералом или идентификатором |
| Присваивание | <идентификатор> = <выражение>;  Выражением может быть идентификатор, литералом, 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. |
| Объявление функции | <тип данных> function <идентификатор>  (<тип данных> <идентификатор>, …) {…} |
| Возврат из функции | return <идентификатор/литерал>; |
| Вывод данных | write (<идентификатор> | <литерал> ); |
| Вызов функции | <идентификатор функции> (<список параметров>); |

В таблице 1.6 представлены все возможные инструкции, которые могут использоваться в языке LAD-2022.

# 1.13 Операции языка

В языке программирования LAD-2022 можно выполнять арифметические операции, а также операции сравнения, все они представлены в таблице 1.7.

Таблица 1.7 – Приоритетности операций языка программирования LAD-2022

|  |  |  |
| --- | --- | --- |
| Операция | Приоритетность операции | Тип оператора  (унарный, бинарный) |
| Арифметические операции | | |
| () (круглые скобки) | 1 | - |
| \* (умножение)  / (деление)  % (остаток от деления) | 2 | бинарный |
| + (бинарный плюс)  - (бинарный минус) | 3 | бинарный |
| Операции сравнения | | |
| < (сравнение: меньше чем)  > (сравнение: больше чем) | - | бинарный |
| & (сравнение: равно)  ! (сравнение: не равно)  @(меньше либо равно)  #(больше либо равно) | - | бинарный |
| Другое | | |
| = (присваивание) | 1 | бинарный |

Максимальным значением приоритетности является “1”, минимальным “3”, операции сравнения приоритета не имеют.

# 1.14 Выражения и их вычисления

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 польской записи для удобства дальнейшего вычисления выражения на языке ассемблер.

# 1.15 Программные конструкции языка

Язык LAD-2022 предполагает наличие программных конструкций, описанных в таблице 1.8.

Таблица 1.8 – Программные конструкции языка LAD-2022

|  |  |
| --- | --- |
| Называние конструкции | Структура конструкции |
| Главная функция | main  {  …  }; |
| Функции | <тип данных> function <идентификатор> ( <тип данных>  <идентификатор>, …)  {  …  return <идентификатор>/<литерал>;  }; |
| Условная конструкция | reque $ <идентификатор1> <оператор> <идентификатор2> $  ?  yes { … }  no { … }  ?  <идентификатор1>, <идентификатор2> - идентификаторы или литералы целочисленного или строкового типа. <оператор> – один из операторов сравнения ( > < & ! @ #), устанавливающий отношение между двумя операндами и организующий условие данной конструкции. При истинности условия выполняется код внутри блока yes, иначе – код внутри блока no. Любой из блоков yes, no может отсутствовать, но не оба блока одновременно. При отсутствии одного из блоков, в зависимости от истинности или ложности условия программа может как выполнить один из заявленных блоков, так и передать управление инструкции, следующей в коде за закрывающим условную конструкцию символом ‘?’. |
| Цикл | reque $ <идентификатор1> <оператор> <идентификатор2> $  ?  cycle { … }  ?  Цикл выполняется, пока истинно условие. |

Таким образом, язык LAD-2022 имеет 4 основные конструкции, которые описаны выше (в таблице 1.8).

# 1.16 Область видимости

Область видимости – «сверху вниз». В языке LAD-2022 требуется обязательное объявление переменной перед её использованием. Все переменные должны находиться внутри программного блока языка. Переменные, объявленные в одной функции, недоступны в другой. Имеется возможность объявления одинаковых переменных в разных блоках кода. После функции main запрещено написание какого-либо кода.

Все идентификаторы являются локальными и обязаны быть объявленными внутри какой-либо функции. Вызов функций в глобальной области видимости запрещен. Объявление пользовательских областей видимости не предусмотрено. Параметры видны только внутри функции, в которой объявлены.

# 1.17 Семантические проверки

Семантические проверки накладывают ограничения на логику написания исходного кода.

Таблица 1.9 – Семантические проверки

|  |  |
| --- | --- |
| № | Правило |
| 1 | Наличие функции main – точки входа в программу; |
| 2 | Единственность точки входа; |
| 3 | Переопределение идентификаторов; |
| 4 | Использование идентификаторов без их объявления; |
| 5 | Проверка соответствия типа функции и возвращаемого параметра; |
| 6 | Проверка соответствия типа переменной и её значения; |
| 7 | Проверка на наличие ключевого слова при объявлении переменной; |
| 8 | Правильность передаваемых в функцию параметров: количество, типы; |
| 9 | Правильность строковых выражений; |
| 10 | Правильность сложных выражений; |
| 11 | Превышение размера строковых и числовых литералов; |
| 12 | Правильность составленного условия цикла/условного оператора; |
| 13 | Проверка правильности деления (деление на ноль) |
| 14 | Проверка на отсутствие кода после главной функции main |

Таблица с перечнем семантических проверок, предусмотренных языком, приведена в таблице 1.9.

# 1.18 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 ассемблер коде переменные имеют глобальную область видимости.

Все переменные размещаются в стеке. Таблица лексем и таблица идентификаторов сохраняются в структуры с выделенной под них динамической памятью, которая очищается по окончанию работы транслятора.

# 1.19 Стандартная библиотека и её состав

Стандартная библиотека написана на языке С++, подключается к транслированному коду на этапе генерации кода. Вызовы стандартных функций доступны там же, где и вызов пользовательских функций.

Таблица 1.10 – Состав стандартной библиотеки

|  |  |  |
| --- | --- | --- |
| Имя функции | Возвращаемое значение | Описание |
| powNumb(uint p, uint n) | uint | Возвращает число, возведённое в степень |
| sqrtNumb(uint a) | uint | Возвращает квадратный корень |
| paused(int x) | int | Приостанавливает выполнение программы |

Также в стандартной библиотеке реализованы функции, которые недоступны конечному пользователю, для манипулирования выводом и проверки в частных случаях деления на 0 и присвоения отрицательного значения беззнаковой переменной. Для вывода предусмотрен оператор write. Эти функции представлены в таблице 1.11.

Таблица 1.11 Дополнительные функции стандартной библиотеки

|  |  |  |
| --- | --- | --- |
| Имя функции | Возвращаемое значение | Описание |
| void printints (int value) | – | Функция для вывода в стандартный поток значения целочисленного идентификатора/литерала. |
| void printstr (char\* str) | – | Функция для вывода в стандартный поток значения строкового идентификатора/литерала. |
| proverkaUint(int numb) | int | Возвращает 1, если параметр не отрицательный, в противном случае возвращает 0 |
| proverkaDel(int numb) | int | Возвращает 1, если параметр не равен 0, в противном случае возвращает 0 |

В данной таблице показаны функции стандартной библиотеке, к которым пользователь не имеет доступ.

# 1.20 Ввод и вывод данных

Ввод данных не поддерживается языком программирования LAD-2022.

Вывод в стандартный поток вывода: write (<идентификатор или литерал>);

# 1.21 Точка входа

В языке LAD-2022 точкой входа является функция main.

Точка входа – это место в исходном коде программы, с которого начинается выполнение программы.

# 1.22 Препроцессор

Препроцессор в языке программирования LAD-2022 не предусмотрен.

# 1.23 Соглашения о вызовах

В языке вызов функций происходит по соглашению о вызовах stdcall.

Особенности stdcall:

– все параметры функции передаются через стек;

– память высвобождает вызываемый код;

– занесение в стек параметров идёт справа налево.

# 1.24 Объектный код

LAD-2022 транслируется в язык Ассемблер.

# 1.25 Классификация сообщений транслятора

В случае возникновения ошибки в коде программы на языке LAD-2022 и выявления её транслятором в текущий файл протокола выводится сообщение. Их классификация сообщений приведена в таблице 1.12.

Таблица 1.12. Классификация сообщений транслятора

|  |  |
| --- | --- |
| Интервал | Описание ошибок |
| 0 – 199 | Системные ошибки, ошибки параметров |
| 200 – 299 | Ошибки лексического анализа |
| 300 – 599 | Ошибки семантического анализа |
| 600 – 900 | Ошибки синтаксического анализа |

В данной таблице показано, как структурированы ошибки по номерам.

# 1.26 Контрольный пример

Контрольный пример демонстрирует главные особенности языка LAD-2022: его фундаментальные типы, основные структуры, функции, использование функций статической библиотеки. Контрольный пример представлен в приложении.

# 2. Структура транслятора

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

Задачей транслятора является преобразование программы, написанной на языке LAD-2022, в программу на языке ассемблер. Составляющими компонентами транслятора являются лексический, синтаксический и семантический анализаторы, а также генератор кода на язык ассемблер.

Исходный код поступает на вход лексического анализатора. Результатом работы последней стадии транслятора является ассемблерный код. Схема транслятора представлена на рисунке 2.1.



Рисунок 2.1 – Структура транслятора

Лексический анализ – это первая фаза трансляции. Результатом работы лексического анализатора является таблица лексем и таблица идентификаторов.

Синтаксический анализ – основная часть транслятора, предназначенная для распознавания синтаксических конструкций языка. Входным параметром синтаксического анализатора является таблица лексем. Синтаксический анализатор выявляет синтаксические ошибки при их наличии и формирует дерево разбора.

Семантический анализ предполагает проверку исходного кода на семантическую согласованность конструкций языка, проверяет правильность текста исходной программы с точки зрения семантики.

Генератор кода – фаза транслятора, выполняющая генерацию ассемблерного кода на основе данных, полученных на предыдущих этапах трансляции. Входными параметрами генератора кода являются таблица идентификаторов и таблица лексем, на основе которых происходит трансляция кода на языке LAD-2022 в код на языке ассемблер.

# 2.2 Перечень входных параметров транслятора

Входные параметры транслятора представлены в таблице 2.1.

Таблица 2.1 – Входные параметры транслятора языка LAD-2022

|  |  |  |
| --- | --- | --- |
| Входной параметр | Описание | Значение по  умолчанию |
| -in:<имя\_файла> | Входной файл с расширением .txt, в котором содержится исходный код на LAD-2022 | Не предусмотрено |
| -log:<имя\_файла> | Будет определять файл, содержащие результат работы программы. | <имя\_in\_файла>.log |
| -out:<имя\_файла> | Файл для записи результата работы транслятора | out.asm |

В данной таблице показано, с какими префиксами используются файлы. Для каждого вида файла свой префикс.

# 2.3 Перечень протоколов, формируемых транслятором и их содержимое

Результатом работы транслятора языка LAD-2022 является исходный код на языке ассемблер и протокол работы транслятора, содержащий основную информацию о процессе обработки исходного кода. Протокол работы транслятора содержит исходный код программы, число символов и строк исходного кода, а также информацию о каждой стадии его работы.

Таблица контроля входных символов представлена на рисунке 3.2.

Таблица 2.2 – Протоколы, формируемые транслятором языка LAD-2022

|  |  |
| --- | --- |
| Протокол | Описание протокола |
| Файл журнала с параметром <log> | 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 параметром <out> | Содержит сгенерированный код на языке Ассемблер. |

Таблица 2.3 предоставляет информацию о перечне протоколов, формируемых транслятором языка LAD-2022 и их назначение.

# 3. Разработка лексического анализатора

# 3.1 Структура лексического анализатора

Лексический анализатор – часть транслятора, выполняющая лексический анализ.

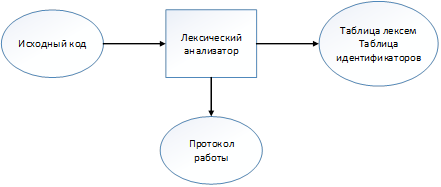


Рисунок 3.1 – структура лексического анализатора

Лексический анализатор преобразует исходный текст, заменяя лексические единицы языка их внутренним представлением – лексемами. Для описания лексики языка программирования применяются регулярные грамматики, относящиеся к типу 3 иерархии Хомского. Язык, заданный регулярной грамматикой, называется регулярным языком (типа 3 иерархии Хомского). Регулярный язык однозначно задается регулярным выражением, а распознавателями для регулярных языков являются конечные автоматы. Грамматики типа 3: GIII = <T,N,P,S> - регулярные грамматики. Регулярные грамматики бывают право линейными и лево линейными.

Правила право линейной грамматики:

A или A, где A,B N, \*.

Правила лево линейной грамматики:

A или A, где A,B N, \*.

Входными данными для лексического анализатора является массив символов, предварительно сформированный при считывании исходного файла. Выходными данными являются таблица лексем, таблица идентификаторов, а также протокол работы. Структура лексического анализатора LAD-2022 представлена на рисунке 3.1.

# 3.2 Контроль входных символов

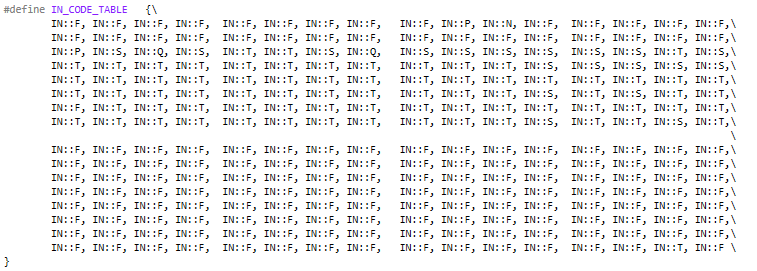
Таблица контроля входных символов представлена на рисунке 3.2

Рисунок 3.2 – Таблица контроля входных символов

Принцип работы таблицы заключается в соответствии значения каждого элемента значению в таблице ASCII.

Описание значения символов: T – разрешённый символ, F – запрещённый символ, I – игнорируемый символ, S – символ-сепаратор, Q – символ идентификации символьных и строковых литералов, P – символы пробела и табуляции N – символ новой строки.

# 3.3 Удаление избыточных символов

Избыточными символами являются символы табуляции и пробелы.

Избыточные символы удаляются на этапе разбиения исходного кода на лексемы.

Описание алгоритма удаления избыточных символов:

* посимвольно считываем файл с исходным кодом программы;
* в отличие от других символов-разделителей, не записываем пробелы и символы табуляции в таблицу лексем;
* продолжаем считывание файла с исходным кодом программы до встречи с лексемой, отличной от пробела или символа табуляции.

# 3.4 Перечень ключевых слов

Лексемы – это символы, соответствующие ключевым словам, символам операций и сепараторам, необходимые для упрощения дальнейшей обработки исходного кода программы. Данное соответствие описано в таблице 3.3.

Таблица 3.3 Соответствие слов, символов операций и сепараторов с лексемами

|  |  |  |
| --- | --- | --- |
| Токен | Лексема | Пояснение |
| int, uint, str, symbol | t | Названия типов данных языка. |

Продолжение таблицы 3.3

|  |  |  |
| --- | --- | --- |
| Токен | Лексема | Пояснение |
| Идентификатор | i | Длина идентификатора – 8 символов. |
| Литерал | l | Литерал любого доступного типа. |
| шестнадцатеричный литерал | h | Литерал в шестнадцатеричном представлении. |
| function | f | Объявление функции. |
| return | r | Выход из функции. |
| main | m | Главная функция. |
| new | n | Объявление переменной. |
| write | p | Ввод данных. |
| yes | w | Истинная ветвь условного оператора. |
| no | e | Ложная ветвь условного оператора. |
| cycle | c | Указывает на начало тела цикла. |
| reque | ~ | Разделение конструкций в цикле/условном операторе. |
| ; | ; | Разделение выражений. |
| , | , | Разделение параметров функций. |
| {} | {} | Открытие и закрытие блока/тела функции. |
| () | () | Открытие и закрытие блока для передачи параметров в функцию, приоритет операций. |
| = | = | Знак присваивания. |
| +  -  \*  /  % | +  -  \*  /  % | Знаки операций. |
| >  <  &  !  @  # | >  <  &  !  @  # | Знаки логических операторов |

Пример реализации таблицы лексем представлен в приложении А.

Также в приложении А находятся конечные автоматы, соответствующие лексемам языка LAD-2022.

# 3.5 Основные структуры данных

Структуры таблиц лексем и идентификаторов данных языка LAD-2022, используемых для хранения, представлены в приложении А.

В таблице лексем содержатся сами лексемы, строка для каждой лексемы, в которой она была замечена. Так же размер самой таблицы лексем В таблице идентификаторов содержится имя идентификатора, его номер в таблице лексем, тип данных, смысловой тип идентификатора и его значение, а также имя родительской функции.

# 3.6 Структура и перечень сообщений лексического анализатора

Перечень сообщений лексического анализатора представлен в таблице 3.4.

Таблица 3.4 Список ошибок лексического анализатора

|  |  |
| --- | --- |
| Код ошибки | Описание ошибки |
| 200 | Недопустимый символ в исходном файле |
| 201 | Неизвестная последовательность символов |
| 202 | Превышен размер таблицы лексем |
| 203 | Превышен размер таблицы идентификаторов |
| 204 – 299 | Зарезервированные ошибки |

Из данной таблицы становится понятна структура прономеровывания ошибок для лексического анализатора

# 3.7 Принцип обработки ошибок

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

# 3.8 Параметры лексического анализатора

Входным параметром лексического анализатора является исходный текст программы, написанный на языке LAD-2022.

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

# 3.9 Алгоритм лексического анализа

Последовательность выполнения алгоритма работы лексического анализатора представлена ниже.

1. Считывание текста из исходного файла в буфер.
2. Далее – посимвольное считывание из буфера. Если текущий символ – конец строки – пункт 8.
3. Проверка символа на допустимость.
4. Если текущий символ сепаратор – запись в таблицу лексем. Если нет – дозапись в строку-буфер до тех пор, пока сепаратор не встретится.
5. Проверка строки-буфера на одно из зарезервированных слов. Если такое есть – запись в таблицу лексем и переход к пункту 2.
6. Проверка строки-буфера на соответствие одному из конечных автоматов. Если нет – пункт 8, выход из программы с ошибкой.
7. Запись в таблицу лексем соответствующего значения. Запись в таблицу идентификаторов того или иного идентификатора или литерала. Переход к пункту 2.
8. Конец работы анализатора.

Все регулярные выражения, имеющиеся в языке, представлены в приложении А.

# 3.10 Контрольный пример

Результат работы лексического анализатора (таблицы лексем и идентификаторов) представлен в приложении А.

# 4. Разработка синтаксического анализатора

# 4.1 Структура синтаксического анализатора

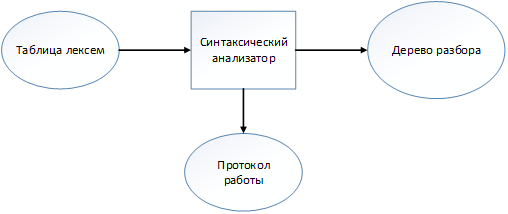
Синтаксический анализ – это фаза трансляции, выполняемая после лексического анализа и предназначенная для распознавания синтаксических конструкций.

Рисунок 4.1 – Структура синтаксического анализатора

Входом для синтаксического анализа является таблица лексем, полученная после фазы лексического анализа. Выходом – дерево разбора. Структура синтаксического анализатора представлена на рисунке 4.1

# 4.2 Контекстно-свободная грамматика, описывающая синтаксис

В синтаксическом анализаторе транслятора языка LAD-2022 используется контекстно-свободная грамматика , где

* T – множество терминальных символов (было описано в разделе 1.2 данной пояснительной записки),
* N – множество нетерминальных символов (первый столбец таблицы 4.1),
* P – множество правил языка (второй столбец таблицы 4.1),
* S – начальный символ грамматики, являющийся нетерминалом.

Эта грамматика имеет нормальную форму Грейбах, она не леворекурсивная 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.

Программное описание правил языка LAD-2022 представлено в приложении Б.

Перечень правил, составляющих грамматику языка, и описание нетермнальных символов LAD-2022 представлены ниже в таблице 4.1.

Таблица 4.1 Таблица правил переходов нетерминальных символов

|  |  |  |
| --- | --- | --- |
| Символ | Правила | Какие правила порождает |
| S | S->tfiPTS  S->pfiPGS  S->m{K} | Стартовые правила, описывающее общую структуру программы |
| P | P->(E)  P->() | Правила для параметров объявляемых функций |
| T | T->{rV;}  T->{KrV;} | Правила для тела функций |
| E | E->ti,E  E->ti | Правила для списка параметров функции |
| F | F->(N)  F->() | Правила для вывозов функций (в т.ч. и в выражениях) |
| N | N->i  N>l  N->l,N  N->i,N | Правила для параметров вызываемых функций |
| R | R->eY?  R>wY?  R>cY?  R->eYwY?  R->wYeY? | Правила составления цикла/условного оператора |
| C | C->iLi  C->iLl  C->lLi  C->lLl | Правила для условия цикла/условного оператора |
| L | L-> >  L-> <  L-> &  L-> !  L-> @  L-> # | Правила для операторов сравнения |
| A | A->+  A->-  A->\*  A->/  A->% | Правила для арифметических |
| V | V->l  V->i  V->h | Правила для простых выражений |

Продолжение таблицы 4.1

|  |  |  |
| --- | --- | --- |
| Символ | Правила | Какие правила порождает |
| W | W->l  W->i  W->h  W->(W)  W->(W)AW  W->iF  W->iAW  W->lAW  W->iFAW | Правила для сложных выражений |
| Y | Y->{X} | Правила для тела цикла/условного выражения |
| K | K->nti=W;K  K->nti;K  K->i=W;K  K->pV;K  K->^;K  K->~$C$?RK  K->iF;K  K->nti=W;  K->nti;  K->i=W;  K->pV;  K->^;  K->~$C$?R  K->iF; | Программные конструкции |
| X | X->i=W;X  X->pV;X  X->^;X  X->iF;X  X->i=W;  X->pV;  X->^;  X->iF; | Программные конструкции внутри цикла/условного оператора |

В данной таблице описаны все правила переходов нетерминальных символов, которые используются в языке LAD-2022.

# 4.3 Построение конечного магазинного автомата

Конечный автомат с магазинной памятью представляет собой семерку, описание которой представлено в таблице 4.2. Структура данного автомата показана в приложении Б.

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 является множеством терминальных и нетерминальных символов, описание которых содержится в разделе 1.2 и в таблице 4.1. |
|  | Алфавит специальных магазинных символов | Алфавит магазинных символов содержит стартовый символ и маркер дна стека |
|  | Функция переходов автомата | Функция представляет из себя множество правил грамматики, описанных в таблице 4.1. |
|  | Начальное состояние автомата | Состояние, которое приобретает автомат в начале своей работы. Представляется в виде стартового правила грамматики (нетерминальный символ S) |
|  | 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 |

В таблице 4.2 представлено описание компонентов магазинного автомата, который используется в языке LAD-2022 для описания правил синтаксиса.

# 4.4 Основные структуры данных

Основные структуры данных синтаксического анализатора включают в себя структуру магазинного автомата и структуру грамматики Грейбах, описывающей правила языка LAD-2022. Данные структуры представлены в приложении Б.

# 4.5 Описание алгоритма синтаксического разбора

Принцип работы конечного автомата с магазинной памятью представлен ниже.

1) В магазин записывается стартовый символ.

2) На основе полученной ранее таблицы лексем формируется входная лента.

3) Запускается автомат.

4) Выбирается цепочка по первому символу, 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 магазина. Иначе возвращаемся в предыдущее сохраненное состояние и выбираем другое правило нетерминала.

6) Если в правиле встретился нетерминал – пункт 4.

7) Если наш символ достиг дна стека, и лента в этот момент пуста, то синтаксический анализ выполнен успешно. Иначе генерируется исключение.

# 4.6 Структура и перечень сообщений синтаксического анализатора

Перечень сообщений синтаксического анализатора представлен в таблице 4.3.

Таблица 4.3 Список ошибок синтаксического анализатора

|  |  |
| --- | --- |
| Код ошибки | Описание ошибки |
| 600 | Неверная структура программы |
| 601 | Не найден список параметров функции |
| 602 | Ошибка в теле функции |
| 604 | Ошибка в списке параметров функции |
| 605 | Ошибка в вызове функции/выражении |
| 606 | Ошибка в списке фактических параметров функции |
| 607 | Ошибка при конструировании цикла/условного выражения |
| 608 | Ошибка в теле цикла/условного выражения |
| 609 | Ошибка в условии цикла/условного выражения |
| 610 | Неверный условный оператор |
| 611 | Неверный арифметический оператор |
| 612 | Неверное выражение. Ожидаются только идентификаторы/литералы |
| 613 | Ошибка в арифметическом выражении |
| 614 | Недопустимая синтаксическая конструкция |
| 615 | Недопустимая синтаксическая конструкция в теле цикла/условного выражения |
| 616-900 | Зарезервированные ошибки |

В данной таблице представлен для ознакомления список ошибок синтаксического анализатора.

# 4.7 Параметры синтаксического анализатора и режимы его работы

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

Выходными параметрами являются трассировка прохода таблицы лексем и правила разбора, которые записываются в файл протокола данного этапа обработки.

**4.8 Принцип обработки ошибок**

Обработка ошибок происходит следующим образом:

* синтаксический анализатор перебирает все правила и цепочки правила грамматики для нахождения подходящего соответствия с конструкцией, представленной в таблице лексем;
* если невозможно подобрать подходящую цепочку, то генерируется соответствующая ошибка, которая записывается в протокол работы и программа останавливается.

# 4.9 Контрольный пример

Пример разбора синтаксическим анализатором исходного кода на языке LAD-2022 представлен в приложении Б. Дерево разбора исходного кода также представлено в приложении Б.

# 5. Разработка семантического анализатора

# 5.1 Структура семантического анализатора

Часть ошибок семантического анализа обрабатываются на этапе лексического анализа. Но ошибки, требующие более сложной обработки (например, несоответствие типов операндов) вынесены в отдельный этап, следующий после синтаксического анализа и преобразования к польской записи.

Рисунок 5.1. Структура семантического анализатора

На вход семантического анализатора подается таблица лексем и таблица идентификаторов, что и показано на рисунке 5.1.

# 5.2 Функции семантического анализатора

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

## 5.3 Перечень сообщений семантического анализатора

Таблица 5.1 – Список ошибок семантического анализатора

|  |  |
| --- | --- |
| Код ошибки | Описание ошибки |
| 300 | Неверно указан тип |
| 301 | Отсутствует точка входа main |
| 302 | Обнаружено несколько точек входа main |
| 304 | В объявлении отсутствует ключевое слово new |
| 305 | Попытка переопределения идентификатора |
| 306 | Превышено максимальное количество параметров функции |
| 308 | Кол-во ожидаемых функцией и передаваемых параметров не совпадают |
| 309 | Несовпадение типов передаваемых параметров |

Продолжение таблицы 5.1

|  |  |
| --- | --- |
| Код ошибки | Описание ошибки |
| 313 | Недопустимый целочисленный литерал |
| 314 | Типы данных в выражении не совпадают |
| 315 | Тип функции и возвращаемого значения не совпадают |
| 316 | Недопустимое строковое выражение справа от знака \'=\' |
| 317 | Недопустимое символьное выражение справа от знака \'=\' |
| 318 | Неверное условное выражение |
| 319 | Деление на ноль |
| 320 | Присвоение значения не входящего в диапазон данного типа |
| 321 | После функции main выполнять какие-либо действия |
| 320-500 | Зарезервированные ошибки |
| 310 | Использование пустого строкового литерала недопустимо |
| 311 | Обнаружен символ \'\"\'. Возможно, не закрыт строковый или символьный литерал |

В таблице 5.1. представлены сообщения, формируемые семантическим анализатором,

# 5.4 Принцип обработки ошибок

Ошибки, возникающие в процессе трансляции программы, фиксируются в протокол, заданный входным параметрами, и командную строку. В случае возникновения ошибок происходит их протоколирование с номером ошибки и диагностическим сообщением. Анализ останавливается после того, как будет найдена, хотя бы одна ошибка, или не найдено ни одной.

# 5.5 Контрольный пример

Соответствие примеров ошибок в исходном коде и диагностических сообщений об ошибках приведено в таблице 8.4.

# 6. Преобразование выражений

# 6.1 Выражения, допускаемые языком

В языке LAD-2022 допускаются выражения, применимые к целочисленным типам данных, а так же выражения условий. В выражениях поддерживаются арифметические операции, такие как +, -, \*, /, % и (), и вызовы функций как операнды арифметических выражений. Для условных выражений те же операции, но с обязательным использованием знаков > , <, &, @, # или !, эти операции не имеют приоритета и записываются в таблицу лексем по мере их встречи в исходном файле.

Таблица 6.1 – Приоритет операций в языке LAD-2022

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

В таблице 6.1. представлен приоритет операций. Самый высокий приоритет имеет значение – 1, самый низкий – 3.

# 6.2 Польская запись

Выражения в языке LAD-2022 преобразовываются к обратной польской записи.

Польская запись – это альтернативный способ записи арифметических выражений, преимущество которого состоит в отсутствии скобок, а также намного более простой обработки выражений впоследствии.

Обратная польская запись – это форма записи математических и логических выражений, в которой операнды расположены перед знаками операций.

Алгоритм построения обратной польской записи представлен ниже.

1. Посимвольно перебираем таблицу лексем.
2. Если текущий символ – какой-либо оператор или идентификатор, записываем его в финальную строку лексем и переходим к пункту 1.
3. Если идентификатор – функция, перед ним в финальную строку записывается токен “@”, означающий вызов функции.
4. Открывающаяся скобка автоматически заносится в стек операций.
5. Закрывающаяся скобка выталкивает все операции из стека в финальную строку и самоуничтожается вместе с открывающейся скобкой.
6. Операция выталкивает все операции с большим или равным приоритетом в финальную строку.
7. Когда встречается символ конца строки кода (“;”) – все операции выталкиваются из стека в финальную строку.
8. В конце операции преобразования к польской нотации после каждой лексемы функции добавляется лексема – число параметров, передаваемых в функцию.

# 6.3 Программная реализация обработки выражений

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

# 6.4 Контрольный пример

Пример преобразования выражения к польской записи представлен в таблице 6.2. Преобразование выражений в формат польской записи в нашем случае необходимо для построения более простых алгоритмов при последующей обработки таблицы лексем.

Таблица 6.2 – Пример преобразования выражения в обратную польскую запись

|  |  |  |
| --- | --- | --- |
| Исходная строка | Результирующая строка | Стек |
| a\*b – c(d) |  |  |
| \*b – c(d) | a |  |
| b – c(d) | a | \* |
| - c(d) | ab | \* |
| c(d) | ab\* | - |
| (d) | ab\*@c | - |
| d) | ab\*@c | - |
| ) | ab\*@cd | - |
|  | ab\*@cd- |  |

В приложении В приведена изменённая таблица лексем, отображающая результаты преобразования выражений в польский формат.

# 7. Генерация кода

# 7.1 Структура генератора кода

Генерация объектного кода — это перевод компилятором внутреннего представления исходной программы в цепочку символов выходного языка.

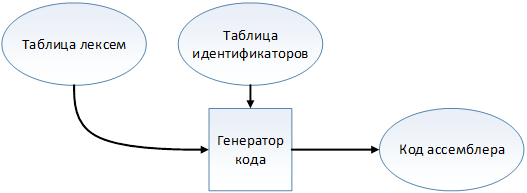


Рисунок 7.1 Структура генератора кода

На вход генератора подаются таблицы лексем и идентификаторов, на основе которых генерируется файл с ассемблерным кодом (Рисунок 7.1).

# 7.2 Представление типов данных в оперативной памяти

Элементы таблицы идентификаторов расположены в разных сегментах языка ассемблер – .data и .const. Идентификаторы языка LAD-2022 размещены в сегменте данных(.data). Литералы – в сегменте констант (const).

Таблица 7.1 – Соответствия типов языков LAD-2022 и Ассемблер

|  |  |  |
| --- | --- | --- |
| Тип идентификатора на языке LAD-2022 | Тип идентификатора на языке ассемблер | Пояснение |
| int | DWORD | Хранит целочисленный тип данных со знаком. |
| uint | DWORD | Хранит целочисленный тип данных без знака. |
| symbol | BYTE | Хранит код символа в кодировке ASCII |
| str | BYTE – для строковых литералов | Литерал хранит последовательность байтов. Переменные хранят указатель на начало строки литерала. |

В данной таблице представлены Соответствия между типами данных идентификаторов на языке LAD-2022 и на языке ассемблер.

# 7.3 Особенности алгоритма генерации кода

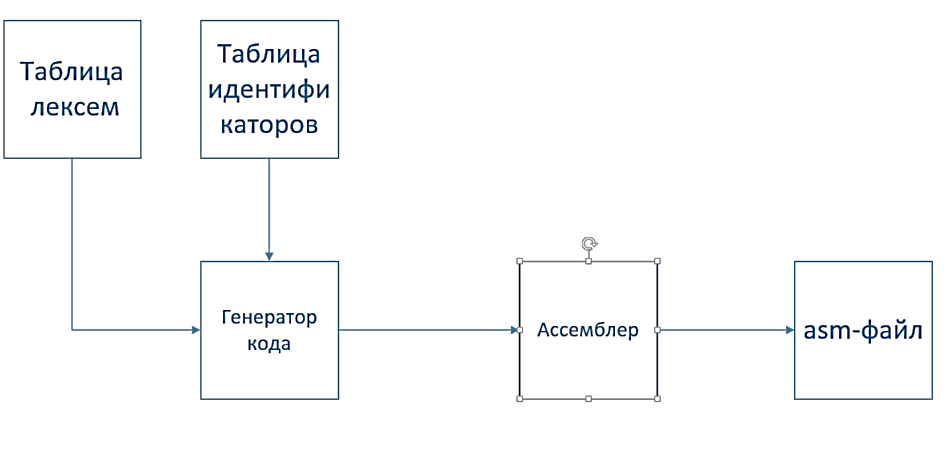
В языке LAD-2022 генерация кода строится на основе таблиц лексем и идентификаторов. Общая схема работы генератора кода представлена на рисунке

Рисунок 7.2 – Структура генератора кода

На вход генератору кода поступают таблицы лексем и идентификаторов исходного код программы на языке LAD-2022. Результаты работы генератора кода выводятся в файл с расширением .asm.

# 7.4 Контрольный пример

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

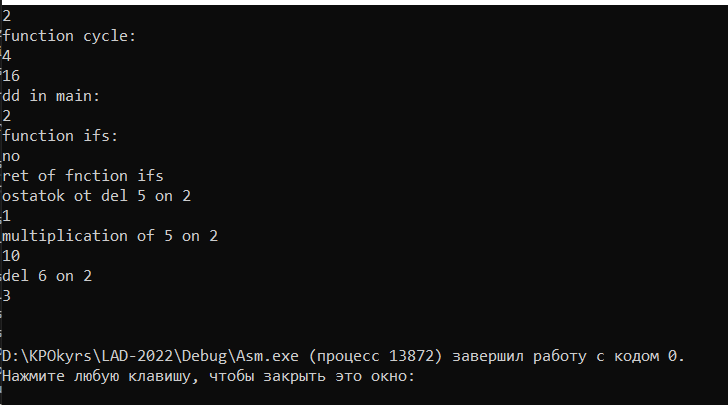


Рисунок 7.2 Результат работы программы на языке LAD-2022

# 8. Тестирование транслятора

# 8.1 Проверка на допустимость символов

В языке LAD-2022 не разрешается использовать запрещённые входным алфавитом символы.

Таблица 8.1 - Тестирование проверки на допустимость символов

|  |  |
| --- | --- |
| Исходный код | Диагностическое сообщение |
| int fu№ion min(int x, int y) {…} | Ошибка 200: Лексическая ошибка: Недопустимый символ в исходном файле(-in), строка 1 |
| str function find(str string)  {  new str копье;  } | Ошибка 200: Лексическая ошибка: Недопустимый символ в исходном файле(-in), строка 3, |

В таблице 8.1 продемонстрирован результат использования запрещённого символа, в примере 1 показан запрещенный символ №, а в примере 2 запрещенными символами являются символы русского алфавита.

# 8.2 Тестирование лексического анализатора

На этапе лексического анализа в языке LAD-2022 могут возникнуть ошибки, описанные в пункте [3.7](#_3.7_Принцип_обработки). Результаты тестирования лексического анализатора показаны в таблице 8.2.

Таблица 8.2 - Тестирование лексического анализатора

|  |  |
| --- | --- |
| Исходный код | Диагностическое сообщение |
| ma@in  {…} | Ошибка 201: Лексическая ошибка: Неизвестная последовательность символов, строка 1 |

Данная ошибка была вызвана тем, что в имени функции main содержался сепаратор, что не позволяется языком LAD-2022.

# 8.3 Тестирование синтаксического анализатора

На этапе синтаксического анализа в языке LAD-2022 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, (строка \_x) |
| int function fi({}) | Ошибка 604: Синтаксическая ошибка: Ошибка в списке параметров функции |
| int function fi(int x) {…}  main { fi(2, ); } | Ошибка 606: Синтаксическая ошибка: Ошибка в списке фактических параметров функции |
| Исходный код | Диагностическое сообщение, (строка \_x) |
| reque $ x > 2 $ ? ? | Ошибка 607: Синтаксическая ошибка: Ошибка при конструировании цикла/условного выражения |
| reque $ ab ! 2 $ | Ошибка 610: Синтаксическая ошибка: Неверный условный оператор |
| write pow(2,2); | Ошибка 612: Синтаксическая ошибка: Неверное выражение. Ожидаются только идентификаторы/литералы |
| new int x;  x = x + + x; | Ошибка 613: Синтаксическая ошибка: Ошибка в арифметическом выражении |
| 2; | Ошибка 614: Синтаксическая ошибка: Отсутствие идентификатора |
| reque $ a < 3 $ ?  cycle {;}  ? | Ошибка 615: Синтаксическая ошибка: Недопустимая синтаксическая конструкция в теле цикла/условного выражения |

Все синтаксические ошибки были успешно обработаны, что видно из таблицы 8.3.

# 8.4 Тестирование семантического анализатора

Семантический анализ в языке LAD-2022 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, (строка \_x) |
| int function fi(){…} | Ошибка 301: Семантическая ошибка: Отсутствует точка входа main |
| main{}  main{} | Ошибка 302: Семантическая ошибка: Обнаружено несколько точек входа main |

Продолжение таблицы 8.4

|  |  |
| --- | --- |
| Исходный код | Диагностическое сообщение, (строка \_x) |
| str function fi(int x)  {…}  main { fi("a");} | Ошибка 309: Семантическая ошибка: Несовпадение типов передаваемых параметров |
| main{new str x="";} | Ошибка 310: Семантическая ошибка: Использование пустого строкового литерала недопустимо |
| main{new str s = “a;} | Ошибка 311: Семантическая ошибка: Обнаружен символ '"'. Возможно, не закрыт строковый литерал |
| new int x=99999999999; | Ошибка 313: Семантическая ошибка: Недопустимый целочисленный литерал |
| new int x;  x = 5 + "abc"; | Ошибка 314: Семантическая ошибка: Типы данных в выражении не совпадают |
| str function fi(){return 5;} | Ошибка 315: Семантическая ошибка: Тип функции и возвращаемого значения не совпадают |
| new str x;  x = "abc" + "d"; | Ошибка 316: Семантическая ошибка: Недопустимое строковое выражение справа от знака '=' |
| a = a/0; | Ошибка 318: Семантическая ошибка: Деление на ноль |
| new uint a = -22; | Ошибка 319: Семантическая ошибка: Присвоение типу uint отрицательного значения |
| int function fi(){…}  main{fi(2,2)} | Ошибка 308: Семантическая ошибка: Кол-во ожидаемых функцией и передаваемых параметров не совпадают |
| main{a = 1;} | Ошибка 304: Семантическая ошибка: В объявлении отсутствует ключевое слово new |
| main {  new int t;  new str t;  } | Ошибка 305: Семантическая ошибка: Попытка переопределения идентификатора |
| int function fi(int x, int y, int z, int s) | Ошибка 306: Семантическая ошибка: Превышено максимальное количество параметров функции |

Все семантические ошибки были успешно обработаны, что указано в данной таблице.

# Заключение

В данном курсовом проекте реализованы поставленные минимальные требования, а также дополнен определённый функционал исходного результата работы. Основной целью курсовой работы было понять принцип работы языков программирования и усвоить эти знания на собственном примере. В ходе работы было изучено много нового, а также закреплены знания, которые были получены ранее. Данный курсовой проект позволил углубиться в изучение таких языков как С++ и ассемблер. При написании приложения были усвоены такие понятия как синтаксический, лексический и семантический анализатор и многие другие.

В итоге был получен примитивный язык программирования LAD-2022, содержащий простые конструкции, позволяющие выполнять несложные математические операции.

Окончательная версия языка LAD-2022 включает:

* 4 типа данных;
* поддержку операции вывода;
* возможность вызова функций стандартной библиотеки;
* наличие 5 арифметических операторов для вычисления выражений;
* наличие операций сравнения;
* структурированную систему для обработки ошибок пользователя.

Основные характеристики транслятора LAD-2022:

* возможность обработки до 3 входных параметров;
* возможность обработки 47 ошибок;
* реализация 21 конечных автоматов;
* реализация 70 цепочек правил грамматики;
* наличие более 3000 строк кода.

Полученные знания при выполнении курсового проекта будут способствовать последующему изучению новых технологий, так как изучение новых языков программирования уже будет проходить на уровне понимания работы языка, а не просто изучение синтаксиса.

# Список использованных источников

1. Курс лекций по КПО Наркевич А.С.
2. Кодировка ASCII > Таблица символов 1251 (ANSI, WIN) [Электронный ресурс]; Режим доступа: <https://tools.otzyvmarketing.ru/blog/poleznoe/Osnova-osnov-kodirovka-ASCII-i-ee-sovremennye-interpretacii->
3. Макросредства языка Ассемблер [Электронный ресурс]; Режим доступа: http://mf.grsu.by/UchProc/livak/arxiv\_22102010/kursi/programming/lections/assm/macros.htm
4. Парадигмы программирования [Электронный ресурс]; Режим доступа: <http://progopedia.ru/paradigm/>
5. Язык программирования. Языки низкого и высокого уровня [Электронный ресурс]; Режим доступа: https://unetway.com/blog/azyk-programmirovania-azyki-nizkogo-i-vysokogo-urovna

# Приложения

# Контрольный пример

Содержимое файла in.txt

|  |
| --- |
| int function cicle(int b)  { new uint d = 2;  reque $ b > 2 $ ?  cycle{  d = pow(d,2);  write d;  b = b - 1;} ?  return 2;}  str function ifs(str b, str c)  { reque $ b # c $?  yes{write "yes";}  no{ write "no";}?  return "ret of fnction ifs";}  main  {  new uint d = 4;  new int dd = 0;  d = sqrt(4);  write d;  write "function cycle:";  dd = cicle(4);  write "dd in main:";  write dd;  write "function ifs:";  new str string = ifs("a","b");  write string;  d = 5 % 2;  write "ostatok ot del 5 on 2";  write d;  write "multiplication of 5 on 2";  d = 5 \* 2;  write d;  write "del 6 on 2";  d = 6 / 2;  write d;  write " ";} |

## Приложение А

Таблица лексем

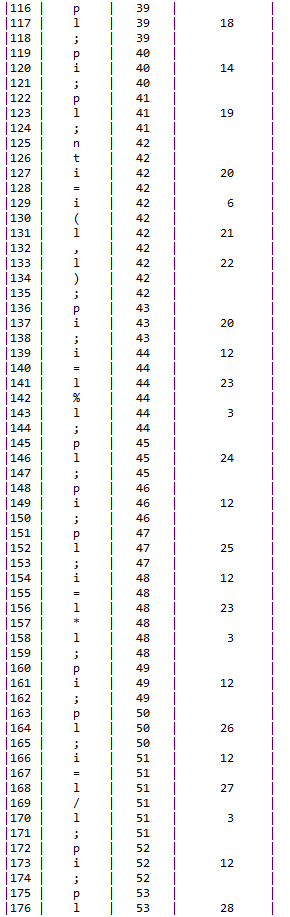
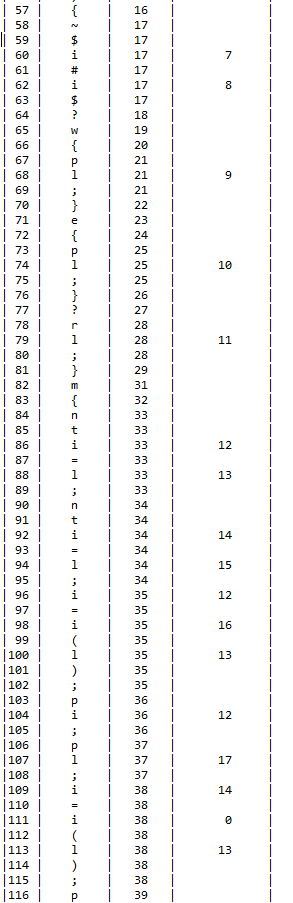
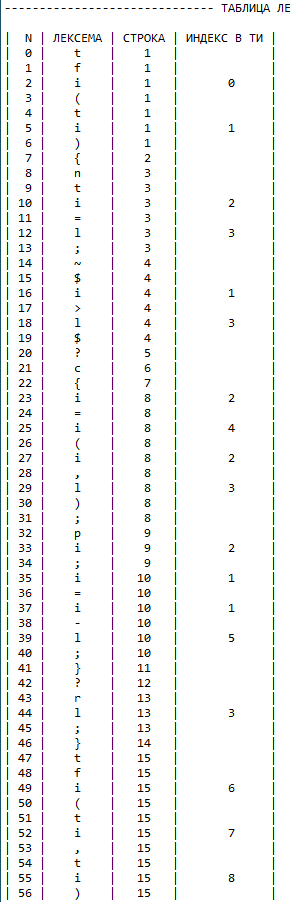
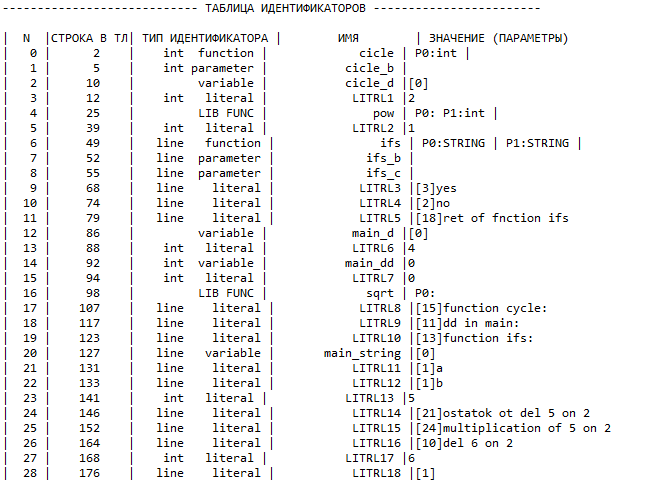


Таблица идентификаторов



Экземпляры конечных автоматов для распознавания лексических конструкций языка

|  |
| --- |
| #pragma once  #include "stdafx.h"  #define N\_GRAPHS 21  #define GRAPH\_SEPARATORS 2,\  FST::NODE(22,\  FST::RELATION(';',1), FST::RELATION('=',1),\  FST::RELATION(',',1), FST::RELATION('[',1),\  FST::RELATION(']',1), FST::RELATION('(',1),\  FST::RELATION(')',1), FST::RELATION('\*',1),\  FST::RELATION('+',1), FST::RELATION('-',1),\  FST::RELATION('?',1), FST::RELATION('/',1),\  FST::RELATION('<',1), FST::RELATION('>',1),\  FST::RELATION('{',1), FST::RELATION('}',1),\  FST::RELATION('&',1), FST::RELATION('!',1),\  FST::RELATION('@',1), FST::RELATION('#',1),\  FST::RELATION('%',1), FST::RELATION('$',1)),\  FST::NODE()  #define GRAPH\_ID 2,\  FST::NODE(26,FST::RELATION('a',1), FST::RELATION('b',1), \  FST::RELATION('c', 1), FST::RELATION('d', 1), \  FST::RELATION('e', 1), FST::RELATION('f', 1), \  FST::RELATION('g', 1), FST::RELATION('h', 1), \  FST::RELATION('i', 1), FST::RELATION('j', 1), \  FST::RELATION('k', 1), FST::RELATION('l', 1), \  FST::RELATION('m', 1), FST::RELATION('n', 1), \  FST::RELATION('o', 1), FST::RELATION('p', 1), \  FST::RELATION('q', 1), FST::RELATION('r', 1), \  FST::RELATION('s', 1), FST::RELATION('t', 1), \  FST::RELATION('u', 1), FST::RELATION('v', 1), \  FST::RELATION('w', 1), FST::RELATION('x', 1), \  FST::RELATION('y', 1), FST::RELATION('z', 1)), \  FST::NODE(34, FST::RELATION('a',1), FST::RELATION('b',1), \  FST::RELATION('c', 1), FST::RELATION('d', 1), \  FST::RELATION('e', 1), FST::RELATION('f', 1), \  FST::RELATION('g', 1), FST::RELATION('h', 1), \  FST::RELATION('i', 1), FST::RELATION('j', 1), \  FST::RELATION('k', 1), FST::RELATION('l', 1), \  FST::RELATION('m', 1), FST::RELATION('n', 1), \  FST::RELATION('o', 1), FST::RELATION('p', 1), \  FST::RELATION('q', 1), FST::RELATION('r', 1), \  FST::RELATION('s', 1), FST::RELATION('t', 1), \  FST::RELATION('u', 1), FST::RELATION('v', 1), \  FST::RELATION('w', 1), FST::RELATION('x', 1), \  FST::RELATION('y', 1), FST::RELATION('z', 1), \  FST::RELATION('0', 1), FST::RELATION('1', 1), \  FST::RELATION('2', 1), FST::RELATION('3', 1), \  FST::RELATION('4', 1), FST::RELATION('5', 1), \  FST::RELATION('6', 1), FST::RELATION('7', 1))  #define GRAPH\_STRING\_LITERAL 3,\  FST::NODE(1, FST::RELATION('\"', 1)),\  FST::NODE(88, \  FST::RELATION('a', 1), FST::RELATION('b', 1), FST::RELATION('c', 1), FST::RELATION('d', 1),\  FST::RELATION('e', 1), FST::RELATION('f', 1), FST::RELATION('g', 1), FST::RELATION('h', 1),\  FST::RELATION('i', 1), FST::RELATION('j', 1), FST::RELATION('k', 1), FST::RELATION('l', 1),\  FST::RELATION('m', 1), FST::RELATION('n', 1), FST::RELATION('o', 1), FST::RELATION('p', 1),\  FST::RELATION('q', 1), FST::RELATION('r', 1), FST::RELATION('s', 1), FST::RELATION('t', 1),\  FST::RELATION('u', 1), FST::RELATION('v', 1), FST::RELATION('w', 1), FST::RELATION('x', 1),\  FST::RELATION('y', 1), FST::RELATION('z', 1),\  FST::RELATION('A', 1), FST::RELATION('B', 1), FST::RELATION('C', 1), FST::RELATION('D', 1),\  FST::RELATION('E', 1), FST::RELATION('F', 1), FST::RELATION('G', 1), FST::RELATION('H', 1),\  FST::RELATION('I', 1), FST::RELATION('J', 1), FST::RELATION('K', 1), FST::RELATION('L', 1),\  FST::RELATION('M', 1), FST::RELATION('N', 1), FST::RELATION('O', 1), FST::RELATION('P', 1),\  FST::RELATION('Q', 1), FST::RELATION('R', 1), FST::RELATION('S', 1), FST::RELATION('T', 1),\  FST::RELATION('U', 1), FST::RELATION('V', 1), FST::RELATION('W', 1), FST::RELATION('X', 1),\  FST::RELATION('Y', 1), FST::RELATION('Z', 1),\  FST::RELATION('0', 1), FST::RELATION('1', 1), FST::RELATION('2', 1), FST::RELATION('3', 1),\  FST::RELATION('4', 1), FST::RELATION('5', 1), FST::RELATION('6', 1), FST::RELATION('7', 1),\  FST::RELATION('8', 1), FST::RELATION('9', 1),\  FST::RELATION(' ', 1), FST::RELATION(',', 1), FST::RELATION('.', 1), FST::RELATION(';', 1),\  FST::RELATION('-', 1), FST::RELATION('+', 1), FST::RELATION('\*', 1), FST::RELATION('/', 1),\  FST::RELATION('=', 1), FST::RELATION(':', 1), FST::RELATION(')', 1), FST::RELATION('(', 1),\  FST::RELATION('}', 1), FST::RELATION('{', 1), FST::RELATION(']', 1), FST::RELATION('[', 1),\  FST::RELATION('!', 1), FST::RELATION('?', 1), FST::RELATION('#', 1), FST::RELATION('&', 1),\  FST::RELATION('>', 1), FST::RELATION('<', 1), FST::RELATION('[', 1), FST::RELATION(']', 1), FST::RELATION('%', 1),\  FST::RELATION('\"', 2)), \  FST::NODE()  #define GRAPH\_CHAR\_LITERAL 3,\  FST::NODE(1, FST::RELATION('\'', 1)),\  FST::NODE(88, \  FST::RELATION('a', 1), FST::RELATION('b', 1), FST::RELATION('c', 1), FST::RELATION('d', 1),\  FST::RELATION('e', 1), FST::RELATION('f', 1), FST::RELATION('g', 1), FST::RELATION('h', 1),\  FST::RELATION('i', 1), FST::RELATION('j', 1), FST::RELATION('k', 1), FST::RELATION('l', 1),\  FST::RELATION('m', 1), FST::RELATION('n', 1), FST::RELATION('o', 1), FST::RELATION('p', 1),\  FST::RELATION('q', 1), FST::RELATION('r', 1), FST::RELATION('s', 1), FST::RELATION('t', 1),\  FST::RELATION('u', 1), FST::RELATION('v', 1), FST::RELATION('w', 1), FST::RELATION('x', 1),\  FST::RELATION('y', 1), FST::RELATION('z', 1),\  FST::RELATION('A', 1), FST::RELATION('B', 1), FST::RELATION('C', 1), FST::RELATION('D', 1),\  FST::RELATION('E', 1), FST::RELATION('F', 1), FST::RELATION('G', 1), FST::RELATION('H', 1),\  FST::RELATION('I', 1), FST::RELATION('J', 1), FST::RELATION('K', 1), FST::RELATION('L', 1),\  FST::RELATION('M', 1), FST::RELATION('N', 1), FST::RELATION('O', 1), FST::RELATION('P', 1),\  FST::RELATION('Q', 1), FST::RELATION('R', 1), FST::RELATION('S', 1), FST::RELATION('T', 1),\  FST::RELATION('U', 1), FST::RELATION('V', 1), FST::RELATION('W', 1), FST::RELATION('X', 1),\  FST::RELATION('Y', 1), FST::RELATION('Z', 1),\  FST::RELATION('0', 1), FST::RELATION('1', 1), FST::RELATION('2', 1), FST::RELATION('3', 1),\  FST::RELATION('4', 1), FST::RELATION('5', 1), FST::RELATION('6', 1), FST::RELATION('7', 1),\  FST::RELATION('8', 1), FST::RELATION('9', 1),\  FST::RELATION(' ', 1), FST::RELATION(',', 1), FST::RELATION('.', 1), FST::RELATION(';', 1),\  FST::RELATION('-', 1), FST::RELATION('+', 1), FST::RELATION('\*', 1), FST::RELATION('/', 1),\  FST::RELATION('=', 1), FST::RELATION(':', 1), FST::RELATION(')', 1), FST::RELATION('(', 1),\  FST::RELATION('}', 1), FST::RELATION('{', 1), FST::RELATION(']', 1), FST::RELATION('[', 1),\  FST::RELATION('!', 1), FST::RELATION('?', 1), FST::RELATION('#', 1), FST::RELATION('&', 1),\  FST::RELATION('>', 1), FST::RELATION('<', 1), FST::RELATION('[', 1), FST::RELATION(']', 1), FST::RELATION('%', 1),\  FST::RELATION('\'', 2)), \  FST::NODE()  #define GRAPH\_INT\_LITERAL 3, \  FST::NODE(11, \  FST::RELATION('-', 1),\  FST::RELATION('0',2),FST::RELATION('1',2),\  FST::RELATION('2',2),FST::RELATION('3',2),\  FST::RELATION('4',2),FST::RELATION('5',2),\  FST::RELATION('6',2),FST::RELATION('7',2),\  FST::RELATION('8',2),FST::RELATION('9',2)),\  FST::NODE(9,\  FST::RELATION('1',2),\  FST::RELATION('2',2),FST::RELATION('3',2),\  FST::RELATION('4',2),FST::RELATION('5',2),\  FST::RELATION('6',2),FST::RELATION('7',2),\  FST::RELATION('8',2),FST::RELATION('9',2)),\  FST::NODE(10,\  FST::RELATION('0',2),FST::RELATION('1',2),\  FST::RELATION('2',2),FST::RELATION('3',2),\  FST::RELATION('4',2),FST::RELATION('5',2),\  FST::RELATION('6',2),FST::RELATION('7',2),\  FST::RELATION('8',2),FST::RELATION('9',2))  #define GRAPH\_ISFALSE 3, \  FST::NODE(1, FST::RELATION('n',1)),\  FST::NODE(1, FST::RELATION('o',2)),\  FST::NODE()  #define GRAPH\_ISTRUE 4, \  FST::NODE(1, FST::RELATION('y',1)),\  FST::NODE(1, FST::RELATION('e',2)),\  FST::NODE(1, FST::RELATION('s',3)),\  FST::NODE()  #define GRAPH\_FUNCTION 9, \  FST::NODE(1, FST::RELATION('f', 1)),\  FST::NODE(1, FST::RELATION('u', 2)),\  FST::NODE(1, FST::RELATION('n', 3)),\  FST::NODE(1, FST::RELATION('c', 4)),\  FST::NODE(1, FST::RELATION('t', 5)),\  FST::NODE(1, FST::RELATION('i', 6)),\  FST::NODE(1, FST::RELATION('o', 7)),\  FST::NODE(1, FST::RELATION('n', 8)),\  FST::NODE()  #define GRAPH\_STATE 6,\  FST::NODE(1,FST::RELATION('r',1)),\  FST::NODE(1,FST::RELATION('e',2)),\  FST::NODE(1,FST::RELATION('q',3)),\  FST::NODE(1,FST::RELATION('u',4)),\  FST::NODE(1,FST::RELATION('e',5)),\  FST::NODE()  #define GRAPH\_UINT 5,\  FST::NODE(1, FST::RELATION('u',1)),\  FST::NODE(1, FST::RELATION('i',2)),\  FST::NODE(1, FST::RELATION('n',3)),\  FST::NODE(1, FST::RELATION('t',4)),\  FST::NODE()  #define GRAPH\_INT 4,\  FST::NODE(1, FST::RELATION('i',1)),\  FST::NODE(1, FST::RELATION('n',2)),\  FST::NODE(1, FST::RELATION('t',3)),\  FST::NODE()  #define GRAPH\_STRING 4, \  FST::NODE(1, FST::RELATION('s',1)),\  FST::NODE(1, FST::RELATION('t',2)),\  FST::NODE(1, FST::RELATION('r',3)),\  FST::NODE()  #define GRAPH\_CHAR 7, \  FST::NODE(1, FST::RELATION('s',1)),\  FST::NODE(1, FST::RELATION('y',2)),\  FST::NODE(1, FST::RELATION('m',3)),\  FST::NODE(1, FST::RELATION('b',4)),\  FST::NODE(1, FST::RELATION('o',5)),\  FST::NODE(1, FST::RELATION('l',6)),\  FST::NODE()  #define GRAPH\_MAIN 5, \  FST::NODE(1,FST::RELATION('m',1)),\  FST::NODE(1,FST::RELATION('a',2)),\  FST::NODE(1,FST::RELATION('i',3)),\  FST::NODE(1,FST::RELATION('n',4)),\  FST::NODE()  #define GRAPH\_CYCLE 6, \  FST::NODE(1,FST::RELATION('c',1)),\  FST::NODE(1,FST::RELATION('y',2)),\  FST::NODE(1,FST::RELATION('c',3)),\  FST::NODE(1,FST::RELATION('l',4)),\  FST::NODE(1,FST::RELATION('e',5)),\  FST::NODE()  #define GRAPH\_WRITE 6, \  FST::NODE(1, FST::RELATION('w',1)),\  FST::NODE(1, FST::RELATION('r',2)),\  FST::NODE(1, FST::RELATION('i',3)),\  FST::NODE(1, FST::RELATION('t',4)),\  FST::NODE(1, FST::RELATION('e',5)),\  FST::NODE()  #define GRAPH\_NEWLINE 5, \  FST::NODE(1, FST::RELATION('e',1)),\  FST::NODE(1, FST::RELATION('n',2)),\  FST::NODE(1, FST::RELATION('d',3)),\  FST::NODE(1, FST::RELATION('l',4)),\  FST::NODE()  #define GRAPH\_RETURN 7, \  FST::NODE(1, FST::RELATION('r',1)),\  FST::NODE(1, FST::RELATION('e',2)),\  FST::NODE(1, FST::RELATION('t',3)),\  FST::NODE(1, FST::RELATION('u',4)),\  FST::NODE(1, FST::RELATION('r',5)),\  FST::NODE(1, FST::RELATION('n',6)),\  FST::NODE()  #define GRAPH\_PROCEDURE 5, \  FST::NODE(1, FST::RELATION('p',1)),\  FST::NODE(1, FST::RELATION('r',2)),\  FST::NODE(1, FST::RELATION('o',3)),\  FST::NODE(1, FST::RELATION('c',4)),\  FST::NODE()  #define GRAPH\_NEW 4,\  FST::NODE(1, FST::RELATION('n',1)),\  FST::NODE(1, FST::RELATION('e',2)),\  FST::NODE(1, FST::RELATION('w',3)),\  FST::NODE()  #define GRAPH\_HEX\_LITERAL 3, \  FST::NODE(1, FST::RELATION('h', 1)),\  FST::NODE(2, FST::RELATION('0', 2), FST::RELATION('1', 2)),\  FST::NODE(22, \  FST::RELATION('0', 2), \  FST::RELATION('1', 2), \  FST::RELATION('2', 2), \  FST::RELATION('3', 2), \  FST::RELATION('4', 2), \  FST::RELATION('5', 2), \  FST::RELATION('6', 2), \  FST::RELATION('7', 2), \  FST::RELATION('8', 2), \  FST::RELATION('9', 2), \  FST::RELATION('A', 2), \  FST::RELATION('B', 2), \  FST::RELATION('C', 2), \  FST::RELATION('D', 2), \  FST::RELATION('E', 2), \  FST::RELATION('F', 2), \  FST::RELATION('a', 2), \  FST::RELATION('b', 2), \  FST::RELATION('c', 2), \  FST::RELATION('d', 2), \  FST::RELATION('e', 2), \  FST::RELATION('f', 2)), \  FST::NODE(22, \  FST::RELATION('0', 2), \  FST::RELATION('1', 2), \  FST::RELATION('2', 2), \  FST::RELATION('3', 2), \  FST::RELATION('4', 2), \  FST::RELATION('5', 2), \  FST::RELATION('6', 2), \  FST::RELATION('7', 2), \  FST::RELATION('8', 2), \  FST::RELATION('9', 2), \  FST::RELATION('A', 2), \  FST::RELATION('B', 2), \  FST::RELATION('C', 2), \  FST::RELATION('D', 2), \  FST::RELATION('E', 2), \  FST::RELATION('F', 2), \  FST::RELATION('a', 2), \  FST::RELATION('b', 2), \  FST::RELATION('c', 2), \  FST::RELATION('d', 2), \  FST::RELATION('e', 2), \  FST::RELATION('f', 2)), \  FST::NODE() |

Программное представление таблицы лексем

|  |
| --- |
| #include "stdafx.h"  #define LEXEMA\_FIXSIZE 1  #define LT\_MAXSIZE 4096  #define LT\_TI\_NULLIDX 0xffffffff  #define LEX\_SEPARATORS 's'  #define LEX\_TYPE 't'  #define LEX\_LITERAL 'l'  #define LEX\_ID 'i'  #define LEX\_FUNCTION 'f'  #define LEX\_MAINFUNC 'm'  #define LEX\_RIGHT ']'  #define LEX\_LEFT '['  #define LEX\_RETURN 'r'  #define LEX\_PRINT 'p'  #define LEX\_SEMICOLON ';'  #define LEX\_COMMA ','  #define LEX\_LEFTBRACE '{'  #define LEX\_RIGHTBRACE '}'  #define LEX\_LEFTHESIS '('  #define LEX\_RIGHTHESIS ')'  #define LEX\_PLUS '+'  #define LEX\_MINUS '-'  #define LEX\_MULTIPLICATION '\*'  #define LEX\_DIRSLASH '/'  #define LEX\_PERSENT '%'  #define LEX\_ASSIGN '='  #define LEX\_DECLARE 'n'  #define LEX\_STATE '~'  #define LEX\_LITERAL\_16 'h'  #define LEX\_ISTRUE 'w'  #define LEX\_ISFALSE 'e'  #define LEX\_NEWLINE '^'  #define LEX\_MORE '>'  #define LEX\_MORE\_AND\_EQAL '#'  #define LEX\_LESS\_AND\_EQAL '@'  #define LEX\_LESS '<'  #define LEX\_EQUAL '&'  #define LEX\_DONT\_EQUAL '!'  #define LEX\_CYCLE 'c'  #define LEX\_PROC 'u'  #define LEX\_DIEZ '?'  namespace LT  {  Entry::Entry()  {  lexema = NULL;  sn = NULL;  idxTI = LT\_TI\_NULLIDX;  }  Entry::Entry(char lexema, int snn, int idxti)  {  this->lexema = lexema;  this->sn = snn;  this->idxTI = idxti;  }  LexTable Create(int size)  {  if (size > LT\_MAXSIZE)  {  throw ERROR\_THROW(202);  }  LexTable\* lexTable = new LexTable;  lexTable->maxsize = size;  lexTable->size = 0;  lexTable->table = new Entry[size];  return \*lexTable;  }  void Add(LexTable& lextable, Entry entry)  {  if (lextable.size + 1 > lextable.maxsize)  throw ERROR\_THROW(202);  lextable.table[lextable.size++] = entry;  }  Entry GetEntry(LexTable& lextable, int n)  {  return lextable.table[n];  }  void Delete(LexTable& lextable)  {  delete[]lextable.table;  }  void writeLexTable(std::ostream\* stream, LT::LexTable& lextable)  {  \*stream << "------------------------------ ТАБЛИЦА ЛЕКСЕМ ------------------------\n" << std::endl;  \*stream << "| N | ЛЕКСЕМА | СТРОКА | ИНДЕКС В ТИ |" << std::endl;  for (int i = 0; i < lextable.size; i++)  {  \*stream << "|" << std::setw(3) << i << " | " << std::setw(4) << lextable.table[i].lexema << " | " << std::setw(3)  << lextable.table[i].sn << " |";  if (lextable.table[i].idxTI == -1)  \*stream << " |" << std::endl;  else  \*stream << std::setw(8) << lextable.table[i].idxTI << " |" << std::endl;  }  }  void writeLexemsOnLines(std::ostream\* stream, LT::LexTable& lextable)  {  \*stream << "\n----------------- ЛЕКСЕМЫ СООТВЕТСТВУЮЩИЕ ИСХОДНОМУ КОДУ ---------------------\n" << std::endl;  for (int i = 0; i < lextable.size; )  {  int line = lextable.table[i].sn;  \*stream << std::setw(3) << line << " | ";  while (lextable.table[i].sn == line)  {  \*stream << lextable.table[i].lexema;  if (lextable.table[i].idxTI != LT\_TI\_NULLIDX)  \*stream << "[" << lextable.table[i].idxTI << "]";  i++;  }  \*stream << std::endl;}  \*stream << "-------------------------------------------------------------------------\n\n";}}; |

## Приложение Б

Грамматика языка LAD-2022

|  |
| --- |
| #define GRB\_ERROR\_SERIES 600  namespace GRB  {  #define NS(n) Rule::Chain::N(n)  #define TS(n) Rule::Chain::T(n)  Greibach greibach(  NS('S'),  TS('$'),  16,  Rule(NS('S'), GRB\_ERROR\_SERIES,  3,  Rule::Chain(6, TS('p'), TS('f'), NS('i'), NS('P'), NS('G'), NS('S')),  Rule::Chain(4, TS('m'), TS('{'), NS('K'),TS('}')),  Rule::Chain(6, TS('t'), TS('f'), TS('i'), NS('P'), NS('T'), NS('S'))  ),  Rule(NS('P'), GRB\_ERROR\_SERIES + 1,  2,  Rule::Chain(3, TS('('), NS('E'), TS(')')),  Rule::Chain(2, TS('('), TS(')'))  ),    Rule(NS('E'), GRB\_ERROR\_SERIES + 4,  2,  Rule::Chain(4, TS('t'), TS('i'), TS(','), NS('E')),  Rule::Chain(2, TS('t'), TS('i'))  ),  Rule(NS('T'), GRB\_ERROR\_SERIES + 2,  2,  Rule::Chain(5, TS('{'), TS('r'), NS('V'), TS(';'), TS('}')),  Rule::Chain(6, TS('{'), NS('K'), TS('r'), NS('V'), TS(';'), TS('}'))  ),  Rule(NS('V'), GRB\_ERROR\_SERIES + 12, 3,  Rule::Chain(1, TS('l')),  Rule::Chain(1, TS('i')),  Rule::Chain(1, TS('h'))  ),  Rule(NS('K'), GRB\_ERROR\_SERIES + 14, 14,  Rule::Chain(7, TS('n'), TS('t'), TS('i'), TS('='), NS('W'), TS(';'), NS('K')),  Rule::Chain(5, TS('n'), TS('t'), TS('i'), TS(';'), NS('K')),  Rule::Chain(5, TS('i'), TS('='), NS('W'), TS(';'), NS('K')),  Rule::Chain(4, TS('p'), NS('V'), TS(';'), NS('K')),  Rule::Chain(3, TS('^'), TS(';'), NS('K')),  Rule::Chain(7, TS('~'), TS('$'), NS('C'), TS('$'), TS('?'), NS('R'), NS('K')),  Rule::Chain(4, TS('i'), NS('F'), TS(';'), NS('K')),  Rule::Chain(6, TS('n'), TS('t'), TS('i'), TS('='), NS('W'), TS(';')),  Rule::Chain(4, TS('i'), TS('='), NS('W'), TS(';')),  Rule::Chain(4, TS('n'), TS('t'), TS('i'), TS(';')),  Rule::Chain(3, TS('p'), NS('V'), TS(';')),  Rule::Chain(2, TS('^'), TS(';')),  Rule::Chain(6, TS('~'), TS('$'), NS('C'), TS('$'), TS('?'), NS('R')),  Rule::Chain(3, TS('i'), NS('F'), TS(';'))  ),  Rule(NS('W'), GRB\_ERROR\_SERIES + 13, 9,  Rule::Chain(1, TS('i')),  Rule::Chain(1, TS('h')),  Rule::Chain(1, TS('l')),  Rule::Chain(3, TS('('), NS('W'), TS(')')),  Rule::Chain(5, TS('('), NS('W'), TS(')'), NS('A'), NS('W')),  Rule::Chain(2, TS('i'), NS('F')),  Rule::Chain(3, TS('i'), NS('A'), NS('W')),  Rule::Chain(3, TS('l'), NS('A'), NS('W')),  Rule::Chain(4, TS('i'), NS('F'), NS('A'), NS('W'))  ),  Rule(NS('A'), GRB\_ERROR\_SERIES + 11, 5,  Rule::Chain(1, TS('+')),  Rule::Chain(1, TS('-')),  Rule::Chain(1, TS('\*')),  Rule::Chain(1, TS('/')),  Rule::Chain(1, TS('%'))  ),  Rule(NS('F'), GRB\_ERROR\_SERIES + 5, 2,  Rule::Chain(3, TS('('), NS('N'), TS(')')),  Rule::Chain(2, TS('('), TS(')'))  ),  Rule(NS('N'), GRB\_ERROR\_SERIES + 6, 4,  Rule::Chain(1, TS('i')),  Rule::Chain(1, TS('l')),  Rule::Chain(3, TS('i'), TS(','), NS('N')),  Rule::Chain(3, TS('l'), TS(','), NS('N'))  ),  Rule(NS('C'), GRB\_ERROR\_SERIES + 9, 4,  Rule::Chain(3, TS('i'), NS('L'), TS('i')),  Rule::Chain(3, TS('i'), NS('L'), TS('l')),  Rule::Chain(3, TS('l'), NS('L'), TS('i')),  Rule::Chain(3, TS('l'), NS('L'), TS('l'))  ),  Rule(NS('L'), GRB\_ERROR\_SERIES + 10, 6,  Rule::Chain(1, TS('<')),  Rule::Chain(1, TS('>')),  Rule::Chain(1, TS('&')),  Rule::Chain(1, TS('#')),  Rule::Chain(1, TS('@')),  Rule::Chain(1, TS('!'))  ),  Rule(NS('R'), GRB\_ERROR\_SERIES + 7, 5,  Rule::Chain(3, TS('e'), NS('Y'), TS('?')),  Rule::Chain(3, TS('w'), NS('Y'), TS('?')),  Rule::Chain(3, TS('c'), NS('Y'), TS('?')),  Rule::Chain(5, TS('e'), NS('Y'), TS('w'), NS('Y'), TS('?')),  Rule::Chain(5, TS('w'), NS('Y'), TS('e'), NS('Y'), TS('?'))  ),  Rule(NS('Y'), GRB\_ERROR\_SERIES + 8, 1,  Rule::Chain(3, TS('{'), NS('X'), TS('}'))  ),  Rule(NS('X'), GRB\_ERROR\_SERIES + 15, 8,  Rule::Chain(5, TS('i'), TS('='), NS('W'), TS(';'), NS('X')),  Rule::Chain(4, TS('p'), NS('V'), TS(';'), NS('X')),  Rule::Chain(3, TS('^'), TS(';'), NS('X')),  Rule::Chain(4, TS('i'), NS('F'), TS(';'), NS('X')),  Rule::Chain(4, TS('i'), TS('='), NS('W'), TS(';')),  Rule::Chain(3, TS('p'), NS('V'), TS(';')),  Rule::Chain(2, TS('^'), TS(';')),  Rule::Chain(3, TS('i'), NS('F'), TS(';'))  )  );  Rule::Chain::Chain(short psize, GRBALPHABET s, ...)  {  nt = new GRBALPHABET[size = psize];  int\* p = (int\*)&s;  for (short i = 0; i < psize; ++i)  {nt[i] = (GRBALPHABET)p[i];}};  Rule::Rule(GRBALPHABET pnn, int piderror, short psize, Chain c, ...)  {  nn = pnn;  iderror = piderror;  chains = new Chain[size = psize];  Chain\* p = &c;  for (int i = 0; i < size; i++) chains[i] = p[i];  };  Greibach::Greibach(GRBALPHABET pstartN, GRBALPHABET pstbottom, short psize, Rule r, ...)  {  startN = pstartN;  stbottomT = pstbottom;  rules = new Rule[size = psize];  Rule\* p = &r;  for (int i = 0; i < size; i++) rules[i] = p[i];  };  Greibach getGreibach()  {  return greibach;  };  short Greibach::getRule(GRBALPHABET pnn, Rule& prule)  {  short rc = -1;  short k = 0;  while (k < size && rules[k].nn != pnn)  {k++;}  if (k < size) prule = rules[rc = k];  return rc;  };  Rule Greibach::getRule(short n)  {  Rule rc;  if (n < size) rc = rules[n];  return rc;  };  char\* Rule::getCRule(char\* b, short nchain)  {  char bchain[200];  b[0] = Chain::alphabet\_to\_char(nn);  b[1] = '-';  b[2] = '>';  b[3] = 0x00;  chains[nchain].getCChain(bchain);  strcat\_s(b, sizeof(bchain) + 5, bchain);  return b;  };  short Rule::getNextChain(GRBALPHABET t, Rule::Chain& pchain, short j)  {  short rc = -1;  while (j < size && chains[j].nt[0] != t) ++j;  rc = (j < size ? j : -1);  if (rc >= 0) pchain = chains[rc];  return rc;  };  char\* Rule::Chain::getCChain(char\* b)  {  for (int i = 0; i < size; i++) b[i] = Chain::alphabet\_to\_char(nt[i]);  b[size] = 0x00;  return b;  };  }; |

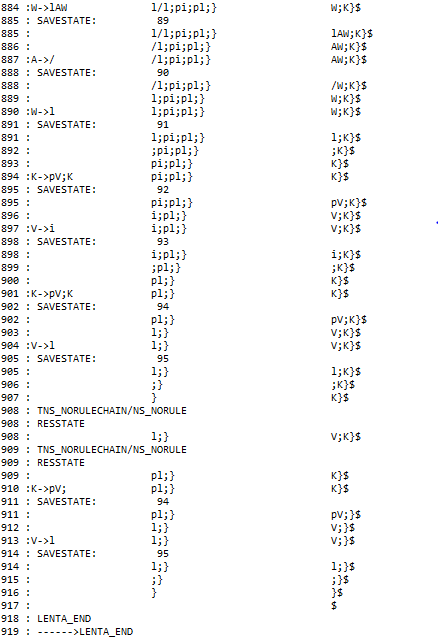
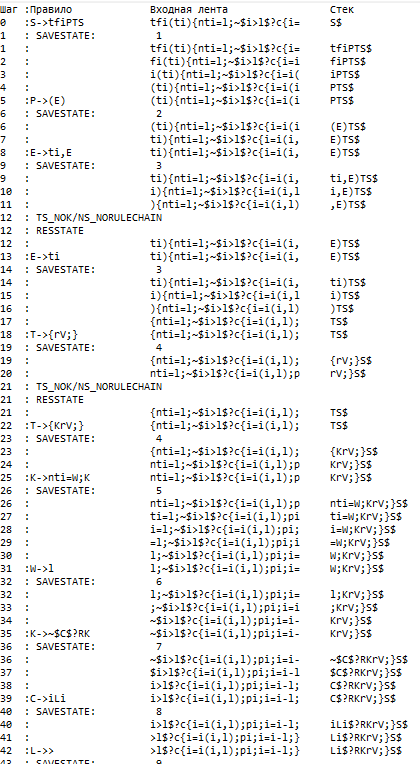
Структура магазинного автомата

|  |
| --- |
| struct Mfst  {  enum RC\_STEP  { NS\_OK,NS\_NORULE,NS\_NORULECHAIN,NS\_ERROR,  TS\_OK,\_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\_digit]; GRBALPHABET\* lenta;  short lenta\_position;  short nrule; short nrulechain; short lenta\_size;  GRB::Greibach grebach;  LexAnalize::LEX lex;  MFSTSTSTACK st;  std::stack<MfstState> storestate;  Mfst();  Mfst(LexAnalize::LEX plex,GRB::Greibach pgrebach);  char\* getCSt(char\* buf);  char\* getCLenta(char\* buf, short pos, short n = 25);  char\* getDiagnosis(short n, char\* buf, const Log::LOG&log);  bool savestate(const Log::LOG& log);  bool reststate(const Log::LOG& log);  bool push\_chain(GRB::Rule::Chain chain);  RC\_STEP step(const Log::LOG& log);  bool start(const Log::LOG& log);  bool savediagnois(RC\_STEP pprc\_step);  void printrules(const Log::LOG& log);  struct Deducation  { short size; short\* nrules; short\* nrulechains;  Deducation() { size = 0; nrules = 0; nrulechains = 0};  } deducation;  bool savededucation();  }; |

Структура грамматики Грейбах

|  |
| --- |
| struct Rule  {  GRBALPHABET nn;  int iderror;  short size;  struct Chain  {  short size;  GRBALPHABET\* nt;  Chain() {size = 0;nt = 0;};  Chain(short psize, GRBALPHABET s, ...);  char\* getCChain(char\* b);  static GRBALPHABET T(char t) {return GRBALPHABET(t);};  static GRBALPHABET N(char n){return -GRBALPHABET(n);};  static bool isT(GRBALPHABET s){return s > 0;};  static bool isN(GRBALPHABET s){return !isT(s);};  static char alphabet\_to\_char(GRBALPHABET s)  {return isT(s) ? char(s) : char(-s);};}\*chains;  Rule() {nn = 0x00;size = 0;}  Rule(GRBALPHABET pnn,int iderror,short psize,Chain c, ...);  char\* getCRule (char\* b,short nchain);  short getNextChain(GRBALPHABET t,  Rule::Chain& pchain,shortj);  };  struct Greibach  {  short size;  GRBALPHABET startN;  GRBALPHABET stbottomT;  Rule\* rules;  Greibach() {short size = 0;startN = 0;stbottomT = 0;rules = 0;};  Greibach(GRBALPHABET pstartN, GRBALPHABET pstbottomT,  short psize,Rule r, ...);  short getRule( GRBALPHABET pnn,Rule& prule);  Rule getRule(short n);};Greibach getGreibach(); |

Разбор исходного кода синтаксическим анализатором

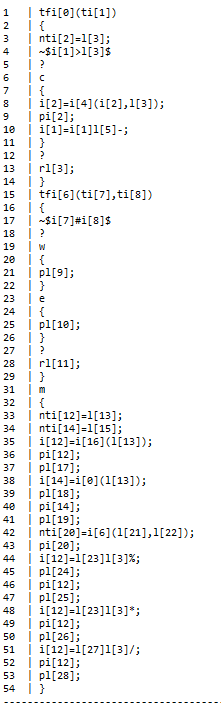


## Приложение В

Программная реализация механизма преобразования в ПОЛИЗ

|  |
| --- |
| int getPriority(LT::Entry& e)  {  switch (e.lexema)  {  case LEX\_LEFTHESIS:  case LEX\_RIGHTHESIS:  return 0;  case LEX\_PLUS:  case LEX\_MINUS:  return 1;  case LEX\_MULTIPLICATION:  case LEX\_DIRSLASH:  case LEX\_PERSENT:  return 2;  case LEX\_LEFT:  case LEX\_RIGHT:  return 3;  default:  return -1;}}  bool PolishNotation(LexAnalize::LEX& tables, Log::LOG& log)  { unsigned curExprBegin = 0; ltVec vec;  LT::LexTable new\_table = LT::Create(tables.lextab.maxsize);  intVec vPosition = getExprPositions(tables);  for (int i = 0; i < tables.lextab.size; i++)  {if (curExprBegin < vPosition.size()  && i == vPosition[curExprBegin])  {int lexCount = fillVector(vPosition[curExprBegin], tables.lextab, vec);  if (lexCount > 1)  {bool rc = setPolishNotation(tables.idtab, log, vPosition[curExprBegin], vec);  if (!rc)  return false;  }  addToTable(new\_table, tables.idtab, vec);  i += lexCount - 1; curExprBegin++; continue;  }  if (tables.lextab.table[i].lexema == LEX\_ID || tables.lextab.table[i].lexema == LEX\_LITERAL)  {  int firstIndef = LexAnalize::getIndexInLT(new\_table, tables.lextab.table[i].idxTI);  if (firstIndef == -1)  firstIndef = new\_table.size;  tables.idtab.table[tables.lextab.table[i].idxTI].idxfirstLE =  firstIndef;}  LT::Add(new\_table, tables.lextab.table[i]);}  tables.lextab = new\_table; return true;}  int fillVector(int posExprBegin, LT::LexTable& lextab, ltVec& vec){  vec.clear();  for (int i = posExprBegin; i < lextab.size; i++)  {  if (lextab.table[i].lexema == LEX\_SEMICOLON)  break;  vec.push\_back(LT::Entry(lextab.table[i]));}  return vec.size();}  void addToTable(LT::LexTable& new\_table, IT::IdTable& idtable, ltVec& vec)  {  for (int i = 0; i < vec.size(); i++)  {  LT::Add(new\_table, vec[i]);  if (vec[i].lexema == LEX\_ID || vec[i].lexema ==LEX\_LITERAL)  {  int firstIndef = LexAnalize::getIndexInLT(new\_table, vec[i].idxTI);  idtable.table[vec[i].idxTI].idxfirstLE = firstIndef;  }  }  }  intVec getExprPositions(LexAnalize::LEX& tables)  {  intVec vec;  bool f\_begin = false;  bool f\_end = false;  int begin = 0;  int end = 0;  for (int i = 0; i < tables.lextab.size; i++)  {  if (tables.lextab.table[i].lexema == LEX\_ASSIGN)  {  begin = i + 1;  f\_begin = true;  continue;  }  if (f\_begin && tables.lextab.table[i].lexema == LEX\_SEMICOLON)  {  end = i;  f\_end = true;  continue;  }  if (f\_begin && f\_end)  {  vec.push\_back(begin);  f\_begin = false;  f\_end = false;  }  }  return vec;  }    bool setPolishNotation(IT::IdTable& idtabl, Log::LOG& log, int lextab\_pos, ltVec& vec)  {  vector<LT::Entry> result;  stack<LT::Entry> st;  bool ignore = false;  for (int i = 0; i < vec.size(); i++)  {  if (ignore)  {  result.push\_back(vec[i]);  if (vec[i].lexema == LEX\_RIGHTHESIS)  {  ignore = false;  }  continue;}  int priority = getPriority(vec[i]);  if (vec[i].lexema == LEX\_LEFTHESIS || vec[i].lexema == LEX\_RIGHTHESIS || vec[i].lexema == LEX\_PLUS || vec[i].lexema == LEX\_MINUS || vec[i].lexema == LEX\_MULTIPLICATION || vec[i].lexema == LEX\_DIRSLASH || vec[i].lexema == LEX\_PERSENT || vec[i].lexema == LEX\_LEFT || vec[i].lexema == LEX\_RIGHT)  {  if (st.empty() || vec[i].lexema == LEX\_LEFTHESIS)  { st.push(vec[i]); continue;}  if (vec[i].lexema == LEX\_RIGHTHESIS)  {  while (!st.empty() && st.top().lexema != LEX\_LEFTHESIS)  {result.push\_back(st.top());  st.pop();}  if (st.top().lexema == LEX\_LEFTHESIS)  {st.pop();}  continue;}  while (!st.empty() && getPriority(st.top()) >= priority)  {  result.push\_back(st.top()); st.pop();}  st.push(vec[i]);}  if (vec[i].lexema == LEX\_LITERAL || vec[i].lexema == LEX\_ID)  {  if (idtabl.table[vec[i].idxTI].idtype == IT::F  || idtabl.table[vec[i].idxTI].idtype == IT::S)  {ignore = true;}  result.push\_back(vec[i]);}  if (vec[i].lexema != LEX\_LEFTHESIS && vec[i].lexema != LEX\_RIGHTHESIS && vec[i].lexema != LEX\_PLUS && vec[i].lexema != LEX\_MINUS && vec[i].lexema != LEX\_MULTIPLICATION && vec[i].lexema != LEX\_DIRSLASH && vec[i].lexema != LEX\_PERSENT && vec[i].lexema != LEX\_ID && vec[i].lexema != LEX\_LITERAL && vec[i].lexema != LEX\_LEFT && vec[i].lexema != LEX\_RIGHT)  {  Log::WriteError(log.stream, Error::geterror(1));  return false;}}  while (!st.empty())  { result.push\_back(st.top());  st.pop();}  vec = result; return true;} |

Результаты преобразования выражений в польский формат



## Приложение Г

Результат генерации кода контрольного примера в Ассемблере

|  |
| --- |
| .586  .model flat, stdcall  includelib libucrt.lib  includelib kernel32.lib  includelib ..\LAD-2022\Debug\StaticLibrary.lib  ExitProcess PROTO : DWORD  .stack 4096  printints PROTO : DWORD  proverkaUint PROTO : DWORD  proverkaDel PROTO : DWORD  printstr PROTO : DWORD  paused PROTO : DWORD, : DWORD  sqrtNumb PROTO : DWORD, : DWORD  powNumb PROTO : DWORD, : DWORD, : DWORD  .CONST  newline byte 13, 10, 0  proverka DWORD 0  LITRL1 dword 2 ;int  LITRL2 dword 1 ;int  LITRL3 byte 'yes', 0 ;str  LITRL4 byte 'no', 0 ;str  LITRL5 byte 'ret of fnction ifs', 0 ;str  LITRL6 dword 4 ;int  LITRL7 dword 0 ;int  LITRL8 byte 'function cycle:', 0 ;str  LITRL9 byte 'dd in main:', 0 ;str  LITRL10 byte 'function ifs:', 0 ;str  LITRL11 byte 'a', 0 ;str  LITRL12 byte 'b', 0 ;str  LITRL13 dword 5 ;int  LITRL14 byte 'ostatok ot del 5 on 2', 0 ;str  LITRL15 byte 'multiplication of 5 on 2', 0 ;str  LITRL16 byte 'del 6 on 2', 0 ;str  LITRL17 dword 6 ;int  LITRL18 byte ' ', 0 ;str  .DATA  temp sdword ?  buffer byte 256 dup(0)  cicle\_d dword 0 ;uint  main\_d dword 0 ;uint  main\_dd dword 0 ;int  main\_string dword ? ;str  .code  cicle PROC,  cicle\_b : sdword  push ebx  push edx  ;-----------  push LITRL1  pop ebx  mov cicle\_d, ebx  push cicle\_d  call proverkaUint  cmp eax, proverka  jz rright0  jnz wwrong0  rright0:  push 0  call ExitProcess  wwrong0:  mov edx, cicle\_b  cmp edx, LITRL1  jg round1  jmp roundnext1  round1:  push LITRL1  push cicle\_d  push offset buffer  call powNumb  push eax  pop ebx  mov cicle\_d, ebx  push cicle\_d  call proverkaUint  cmp eax, proverka  jz rright1  jnz wwrong1  rright1:  push 0  call ExitProcess  wwrong1:  push cicle\_d  call printints  push cicle\_b  push LITRL2  pop ebx  pop eax  sub eax, ebx  push eax  pop ebx  mov cicle\_b, ebx  mov edx , cicle\_b  cmp edx, LITRL1  jg round1  roundnext1:;----- restore registers -----  pop edx  pop ebx  mov eax, LITRL1  ret  cicle ENDP  ifs PROC,  ifs\_b : dword, ifs\_c : dword  ;---------save register ----  push ebx  push edx  ;-----------  mov edx, ifs\_b  cmp edx, ifs\_c  jge right2  jle wrong2  right2:  push offset LITRL3  call printstr  jmp roundnext2  wrong2:  push offset LITRL4  call printstr  roundnext2:  ;----- restore registers -----  pop edx  pop ebx  mov eax, offset LITRL5  ret  ifs ENDP  ;--------------MAIN------------  main PROC  push LITRL6  pop ebx  mov main\_d, ebx  push main\_d  call proverkaUint  cmp eax, proverka  jz rright2  jnz wwrong2  rright2:  push 0  call ExitProcess  wwrong2:  push LITRL7  pop ebx  mov main\_dd, ebx  push LITRL6  push offset buffer  call sqrtNumb  push eax  pop ebx  mov main\_d, ebx  push main\_d  call proverkaUint  cmp eax, proverka  jz rright3  jnz wwrong3  rright3:  push 0  call ExitProcess  wwrong3:  push main\_d  call printints  push offset LITRL8  call printstr  push LITRL6  call cicle  push eax  pop ebx  mov main\_dd, ebx  push offset LITRL9  call printstr  push main\_dd  call printints  push offset LITRL10  call printstr  push offset LITRL12  push offset LITRL11  call ifs  mov main\_string, eax  push main\_string  call printstr  push LITRL13  push LITRL1  pop ebx  pop eax  cdq  mov edx,0  idiv ebx  push edx  pop ebx  mov main\_d, ebx  push main\_d  call proverkaUint  cmp eax, proverka  jz rright4  jnz wwrong4  rright4:  push 0  call ExitProcess  wwrong4:  push offset LITRL14  call printstr  push main\_d  call printints  push offset LITRL15  call printstr  push LITRL13  push LITRL1  pop ebx  pop eax  imul eax, ebx  push eax  pop ebx  mov main\_d, ebx  push main\_d  call proverkaUint  cmp eax, proverka  jz rright5  jnz wwrong5  rright5:  push 0  call ExitProcess  wwrong5:  push main\_d  call printints  push offset LITRL16  call printstr  push LITRL17  push LITRL1  pop ebx  pop eax  cdq  idiv ebx  push eax  pop ebx  mov main\_d, ebx  push main\_d  call proverkaUint  cmp eax, proverka  jz rright6  jnz wwrong6  rright6:  push 0  call ExitProcess  wwrong6:  push main\_d  call printints  push offset LITRL18  call printstr  push 0  call ExitProcess  main ENDP  end main |

# Приложение Е

Графический материал. Граф дерева разбора.