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 DDA-2020»

Выполнил студент Денисюк Дмитрий Александрович

(Ф.И.О.)

Руководитель проекта ст.пр. Наркевич Аделина Сергеевна

(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 ст.пр. Наркевич Аделина Сергеевна

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

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

Нормоконтролер ст.пр. Наркевич Аделина Сергеевна

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

Курсовой проект защищен с оценкой

Минск 2020

**Оглавление**

[Введение 5](#_Toc58683400)

[Глава 1. Спецификация языка программирования 6](#_Toc58683401)

[1.1 Характеристика языка программирования 6](#_Toc58683402)

[1.2 Алфавит языка 6](#_Toc58683403)

[1.3 Применяемые сепараторы 7](#_Toc58683404)

[1.4 Применяемые кодировки 7](#_Toc58683405)

[1.5 Типы данных 7](#_Toc58683406)

[1.6 Преобразование типов данных 8](#_Toc58683407)

[1.7 Идентификаторы 8](#_Toc58683408)

[1.8 Литералы 9](#_Toc58683409)

[1.9 Объявления данных 9](#_Toc58683410)

[1.10 Инициализация данных 9](#_Toc58683411)

[1.11 Инструкции языка 10](#_Toc58683412)

[1.10 Операции языка 10](#_Toc58683413)

[1.12 Выражения и их вычисления 11](#_Toc58683414)

[1.13 Конструкции языка 11](#_Toc58683415)

[1.14 Область видимости идентификаторов 12](#_Toc58683416)

[1.15 Семантические проверки 12](#_Toc58683417)

[1.16 Распределение оперативной памяти на этапе выполнения 13](#_Toc58683418)

[1.17 Стандартная библиотека и её состав 13](#_Toc58683419)

[1.18 Ввод и вывод данных 14](#_Toc58683420)

[1.19 Точка входа 14](#_Toc58683421)

[1.20 Препроцессор 14](#_Toc58683422)

[1.21 Соглашения о вызовах 14](#_Toc58683423)

[1.22 Объектный код 14](#_Toc58683424)

[1.23 Классификация сообщений транслятора 14](#_Toc58683425)

[1.24 Контрольный пример 15](#_Toc58683426)

[Глава 2. Структура транслятора 16](#_Toc58683427)

[2.1 Компоненты транслятора, их назначение и принципы взаимодействия 16](#_Toc58683428)

[2.2 Перечень входных параметров транслятора 17](#_Toc58683429)

[2.3 Протоколы, формируемые транслятором 17](#_Toc58683430)

[Глава 3. Разработка лексического анализатора 18](#_Toc58683431)

[3.1 Структура лексического анализатора 18](#_Toc58683432)

[3.2 Контроль входных символов 18](#_Toc58683433)

[3.3 Удаление избыточных символов 19](#_Toc58683434)

[3.4 Перечень ключевых слов 19](#_Toc58683435)

[3.5 Основные структуры данных 21](#_Toc58683436)

[3.6 Структура и перечень сообщений лексического анализатора 23](#_Toc58683437)

[3.7 Принцип обработки ошибок 23](#_Toc58683438)

[3.8 Параметры лексического анализатора и режим его работы 24](#_Toc58683439)

[3.9 Алгоритм лексического анализа 24](#_Toc58683440)

[3.10 Контрольный пример 24](#_Toc58683441)

[Глава 4. Разработка синтаксического анализатора 25](#_Toc58683442)

[4.1 Структура синтаксического анализатора. 25](#_Toc58683443)

[4.2 Контекстно-свободная грамматика, описывающая синтаксис языка 25](#_Toc58683444)

[4.3 Построение конченого магазинного автомата 26](#_Toc58683445)

[4.4 Основные структуры данных 27](#_Toc58683446)

[4.5 Описание алгоритма синтаксического разбора 28](#_Toc58683447)

[4.6 Структура и перечень сообщений синтаксического анализатора 28](#_Toc58683448)

[4.7 Параметры синтаксического анализатора и режимы его работы 28](#_Toc58683449)

[4.8 Принцип обработки ошибок 29](#_Toc58683450)

[4.9 Контрольный пример 29](#_Toc58683451)

[Глава 5. Разработка семантического анализатора 30](#_Toc58683452)

[5.1 Структура семантического анализатора 30](#_Toc58683454)

[5.2 Функции семантического анализатора 30](#_Toc58683455)

[5.3 Структура и перечень сообщений семантического анализатора 31](#_Toc58683456)

[5.4 Принцип обработки ошибок 32](#_Toc58683457)

[5.5 Контрольный пример 32](#_Toc58683458)

[Глава 6. Преобразование выражений 33](#_Toc58683459)

[6.1 Выражения, допускаемые языком 33](#_Toc58683460)

[6.2 Польская запись и принцип ее построения 34](#_Toc58683461)

[6.3 Программная реализация обработки выражений 34](#_Toc58683462)

[6.4 Контрольный пример 35](#_Toc58683463)

[Глава 7. Генерация кода 36](#_Toc58683464)

[7.1 Структура генератора кода 36](#_Toc58683465)

[7.2 Представление типов данных в памяти 36](#_Toc58683466)

[7.3 Статическая библиотека 37](#_Toc58683467)

[7.4 Особенности алгоритма генерации кода 38](#_Toc58683468)

[7.5 Контрольный пример 38](#_Toc58683469)

[8. Тестирование транслятора 39](#_Toc58683470)

[8.1 Общие положения 39](#_Toc58683471)

[8.2 Результаты тестирования 39](#_Toc58683472)

[Заключение 41](#_Toc58683473)

[Список использованных источников 42](#_Toc58683474)

[Приложение А 43](#_Toc58683475)

[Приложение Б 44](#_Toc58683476)

[Приложение В 46](#_Toc58683477)

[Приложение Г 48](#_Toc58683478)

[Приложение Д 50](#_Toc58683479)

[Приложение Е 52](#_Toc58683480)

# **Введение**

Целью курсового проекта является разработка компилятора для своего языка программирования: DDA-2020. Этот язык программирования предназначен для выполнения простых задач общего назначения.

Компилятор языка DDA-2020 состоит из следующих составных частей:

* лексический анализатор;
* синтаксический анализатор;
* семантический анализатор;
* преобразователя выражений в обратный польский формат;
* генератор исходного кода в язык ассемблера.

Исходя из цели курсового проекта, были определены следующие задачи:

* разбработка спецификации языка программирования;
* разбратка структуры транслятора;
* разработка лексического и семантического анализаторов;
* разработка синтаксического анализатора;
* преобразование выражений в обратный польский формат;
* генерация кода в язык ассемблера;
* автоматический запуск генерируемого кода;
* тестирование транслятора.

Решения каждой из поставленных задач будут приведены в соответствующих главах курсового проекта, а именно:

В первой главе работы определена спецификация языка программирования, т.е. описан синтаксис, семантика языка и возможности языка.

Во второй главе работы представлена общая структура транслятора, т.е. основные компоненты и их назначение.

В третьей главе работы показана разработка лексического анализатора, порождающего таблицу лексем и таблицу идентификаторов.

В четвертой главе работы рассказывается о синтаксическом анализаторе, который выполняет синтаксический разбор текста с распечаткой протокола разбора и дерева разбора на основе таблицы лексем.

В пятой главе описан семантический анализатор, а также говорится об семантических проверках компилятора.

В шестой главе объясняется принцип преобразования выражений в обратный польский формат.

В седьмой главе представлена генерация кода, где из исходного кода на языке DDA-2020, порождается код ассемблера.

В восьмой главе описывается тестирование транслятора, а также показываются различные исключительные ситуации.

# Глава 1. Спецификация языка программирования

* 1. **Характеристика языка программирования**

Язык программирования DDA-2020 – транслируемый, универсальный, процедурный, строго типизированный язык высокого уровня. В языке используется четыре типа данных: целочисленный (number), беззнаковый целочисленный (ubyte), строковый (string) и логический (bool).

* 1. **Алфавит языка**

В основе алфавита DDA-2020 лежит таблица символов ACSII. Разрешены латинские символы, символы кириллицы, символы-сепараторы, символы операций и другие, представленные на рисунке 1.1. и рисунке 1.2.

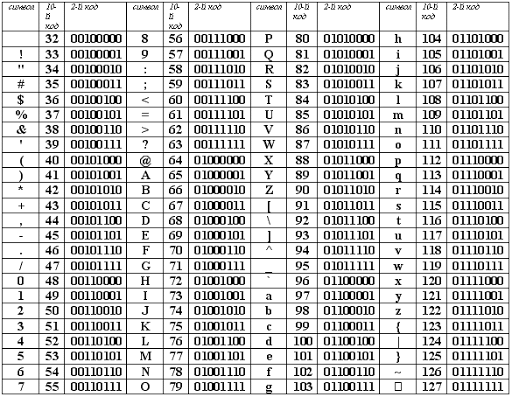


Рисунок 1.1 - Алфавит входных символов

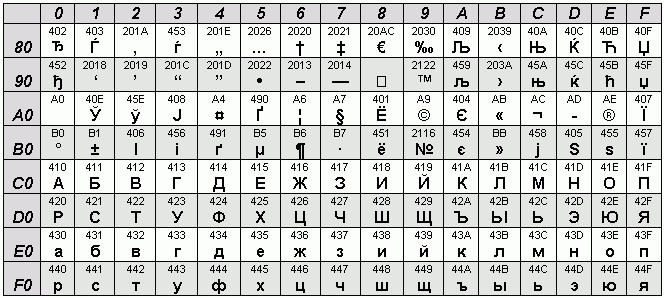


Рисунок 1.2 - Алфавит входных символов

* 1. **Применяемые сепараторы**

Сепараторы необходимы для разделения операций языка. Сепараторы, используемые в языке программирования DDA-2020, приведены в таблице 1.1.

Таблица 1.1 - Применяемые сепараторы

|  |  |
| --- | --- |
| Сепаратор | Назначение сепаратора |
| ; | Разделитель инструкций |
| { } | Программный блок |
| ( ) | Параметры, приоритетность операций |
| пробел | Служит для разделения программных конструкций. Допускается везде, кроме идентификаторов и ключевых слов |
| , | Разделитель параметров в функции |
| and, or | Разделить условных выражений |
| [ ] | Индексация к идентификатору |

* 1. **Применяемые кодировки**

При трансляции применяется стандартная кодировка ACSII. Описание кодировки представлено в пункте 1.2

* 1. **Типы данных**

Пользовательские типы данных не поддерживаются. Допускается использование фундаментальных типов данных, определенных в таблице 1.2.

Таблица 1.2 - Фундаментальные типы данных

|  |  |
| --- | --- |
| Тип данных | Описание |
| number | Целочисленный тип данных. Занимает 4 байта, значение от  –231 до 231-1, автоматическая инициализация 0;  Предусмотрены следующие арифметические операции:  + - бинарная операция суммирования (number + number);  - - бинарная операция вычитания (number - number);  \* - бинарная операция умножения (number \* number);  / - бинарная операция деления (number / number).  Предусмотрены следующие побитовые операции:  & - бинарное операция И (number&number);  | - бинарная операция ИЛИ (number|number);  ~ - унарная операция НЕ (~number). |
| string | Строковый тип данных. Предназначен для хранения и работы с символами. Максимально 255 символов, первый байт – длина строки, автоматическая инициализация строкой с длиной 0  Операции над данными строкового типа: возможно присваивание строковому идентификатору значения другого строкового идентификатора, конкатенация строк, копирование одной строки в другую, вычисление длинны строки. |
| ubyte | Беззнаковый целочисленный тип данных. Занимает 1 байт, значение от 0 до 255, автоматическая инициализация 0;  Предусмотрены следующие арифметические операции:  + - бинарная операция суммирования (ubyte + ubyte);  - - бинарная операция вычитания (ubyte - ubyte);  \* - бинарная операция умножения (ubyte \* ubyte);  / - бинарная операция деления (ubyte / ubyte).  Предусмотрены следующие побитовые операции:  & - бинарное операция И (ubyte & ubyte);  | - бинарная операция ИЛИ (ubyte | ubyte);  ~ - унарная операция НЕ (~ubyte). |
| bool | Логический тип данных. Занимает 1 байт, значение 0 (false) или любое отличное от 0 (true), автоматическая инициализация 0 (false). |

* 1. **Преобразование типов данных**

Преобразование поддерживается только для целочисленных типов данных (number и ubyte). Для остальных типов преобразование не поддерживается.

* 1. **Идентификаторы**

Для именования функций, параметров и переменных используются идентификаторы. Не предусмотрены зарезервированные идентификаторы. Имя идентификаторов не должно совпадать с ключевыми словами языка и с именами функций стандартной библиотеки. Имя идентификатора составляется по следующим образом:

* состоит из символов латинского алфавита [a..zA..Z].
* максимальная длина идентификатора равна 15. При превышении максимально значения длина идентификатора усекается до 15.

Пример правильного объявления:

def bool IsTrue;

Пример неправильного объявления:

def number число;

def ubyte main;

* 1. **Литералы**

Предусмотрены числовые (number), строковые (string), целочисленные беззнаковые (ubyte) и логические (bool) литералы. Правила записи приведены в таблице 1.3.

Таблица 1.3 - Правила записи литералов

|  |  |
| --- | --- |
| Тип литерала | Описание литерала |
| Числовые | Значение от –231 до 231-1. Могут состоять только из цифр [0..9] без дробной части. Если целочисленный литерал начинается с 0, то он идентифицируется, как целочисленное значение в восьмеричном представлении. При выходе за пределы допустимости произойдет переполнение. |
| Строковые | Состоит из символов, заключенных в “ “ (двойные кавычки). Максимальное число символов 255. В случае превышения длины литерала выдается ошибка. |
| Целочисленный беззнаковый | Состоит из символа, заключенного в ‘ ‘ (одинарные кавычки). В кавычках может быть заключен любой допустимый символ языка. |
| Логические | Распознаются при помощи ключевых слов “true” и “false”, соответственно значения от 0 (false) до 1 (true) |

* 1. **Объявления данных**

В языке программирования DDA-2020 переменная должны быть объявлена до ее использования. Областью видимости переменной является блок функции, в которой она определена. Вне блока функции определении функции запрещено. Не допустимо объявление глобальных переменных. Конструкция для объявления переменных:

def <тип данных> <идентификатор>[=<литерал>|<идентификатор>| <выражение>| <вызов функции>];

**1.10 Инициализация данных**

При объявлении переменной допускается инициализация данных. Объектами-инициализаторами могут быть идентификаторы, литералы, выражения и вызов функции. Предусмотрены значения по умолчанию, если переменные не инициализированы: 0 – для целочисленных типов данных, пустая строка (строка размером 0) – для строкового типа данных, 0 (false) – для логического типа данных.

**1.11 Инструкции языка**

В языке программирования DDA-2020 применяются инструкции, представленные в таблице 1.4.

Таблица 1.4 - Инструкции языка

|  |  |
| --- | --- |
| Инструкция языка | Синтаксис |
| Вызов функций | <идентификатор функции> ([<идентификатор> | <литерал> , …]); |
| Возврат из функции | return <идентификатор> | <литерал> | <выражение>; |
| Объявление переменной | def <тип данных> <идентификатор>; |
| Присваивание | <идентификатор> = <литерал> | <выражение> | <идентификатор> | <вызов функции>; |
| Вывод данных | print <идентификатор> | <литерал> | <выражение>; |
| Ввод данных | get <идентификатор>; |
| Инкремент (декремент) | ++ (--) <идентификатор>; |

**1.10 Операции языка**

Операции языка применимы исключительно к логическому и целочисленным типом данным. Для строкового типа операции языка не предусмотрены.

Наибольший приоритет имеют операции заключенные в круглые скобки, затем операции умножения, деления и побитовые операции, затем операции сложения и вычитания. При одинаковом приоритете первой выполнится операция, расположенная левее.

Операции языка DDA-2020 приведены в таблице 1.5.

Таблица 1.5 - Операции языка

|  |  |
| --- | --- |
| Тип | Операторы |
| Арифметические | () – приоритетность операций  + ̶ сложение  -  ̶ вычитание  \* ̶ умножение  / ̶ деление |
| Побитовые | ~ - инверсия  & - конъюнкция  | - дизъюнкция |
| Сравнение | < - меньше  > - больше  == - равно  != - не равно  <= - меньше или равно  >= - больше или равно |

* 1. **Выражения и их вычисления**

Предусмотрены следующие правила составления выражений:

* + выражения читаются слева направо и записываются в одну строку;
  + реализация выражений происходит с помощью обратной польской записи;
  + для изменения приоритета операция используются круглые скобки;
  + в выражениях должен использоваться одинаковый тип данных;
  + выражения могут состоять из разных типов операций, однако в выражении допустима только одна операция сравнения.
  1. **Конструкции языка**

Программные конструкции языка представлены в таблице 1.7.

Таблица 1.7 - Программные конструкции языка

|  |  |
| --- | --- |
| Конструкция | Представление в языке |
| Главная функция | main  {  <инструкции языка>  } |
| Функция | <тип данных> function <идентификатор> ([<тип данных> <идентификатор>, …])  {  <инструкции языка>  } |
| Условный оператор | if (<условие>)  {  <инструкции языка>  }  [elif (<условие>)  {  <инструкции языка>  }]  [else  {  <инструкции языка>  }]  Условия проверяются сверху вниз. Когда обнаруживается истинное условие, то выполняется блок инструкций, следующий за условием, а остальная часть конструкции игнорируется. Если не найдено ни одного истинного условия, то выполняются инструкции в блоке следующим за else. В конструкции может отсутствовать elif и (или) else. |
| Цикл | for <идентификатор> in <идентификатор> | <литерал> .. <идентификатор> | <литерал> step <выражение>  {  <инструкции языка>  }  Идентификатору, следующему за for, присваивается значение идентификатора или литерала следующего за in. После каждой итерации цикла выполняется выражение следующее за step, до того момента, пока исходный идентификатор меньше идентификатора или литерала стоящего перед step. |

Также, если в фигурных скобках заключена только одна инструкция, то возможна эквивалентная замена фигурных скобок на => и следующую за ней инструкцию. Например:

if (<условие>) => <инструкция языка>

* 1. **Область видимости идентификаторов**

Все идентификаторы обязаны быть объявленными внутри функции. Вне функции объявление идентификаторов недопустимы. Глобальных переменных нет, только локальные. Параметры видны только внутри функции, в которой объявлены. Область видимости сверху вниз.

* 1. **Семантические проверки**

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

Таблица 1.8 - Перечень семантических проверок

|  |  |
| --- | --- |
| Номер | Правило |
| 1 | Наличие функции main, как точки входа в программу |
| 2 | Наличие только одной точки входа |
| 3 | Недопустимость переопределения идентификатора |
| 4 | Использование идентификаторов без их объявления |
| 5 | Усечение слишком длинных идентификаторов до 15 символов |
| 6 | Сначала идет проверка на ключевые слова, потом на идентификатор. Не допускается совпадении имени идентификатора с ключевым словом |
| 8 | Правильность передаваемых в функцию параметров: количество, типы |
| 9 | Не допускается совпадения имени функции с именем функций стандартной библиотеки |
| 10 | Усечение слишком длинного значения string-литерала до 255 символов |
| 11 | Использование допустимых типов в выражениях, условиях, цикле |

* 1. **Распределение оперативной памяти на этапе выполнения**

Данные размещаются в стеке. 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.

* 1. **Стандартная библиотека и её состав**

В языке DDA-2020 присутствует стандартная библиотека, которая автоматически подключается при трансляции исходного кода. У каждого типа данных есть свои функции реализующие различные команды. Содержимое библиотеки и описание функций представлено в таблице 1.9.

Таблица 1.9 - Стандартная библиотека языка DDA-2020

|  |  |
| --- | --- |
| Функция | Описание |
| bool strconcat (string a, string b); | Строковая функция. Принимает в качестве параметров 2 строки. Копирует вторую строку к концу первой строки. В случае успеха возвращает true, иначе false |
| bool strcopy (string a, string b); | Строковая функция. Принимает в качестве параметров 2 строки. Копирует содержимое второй строки в первую, возвращает true в случае успешного копирования, иначе false |
| number strlength(string a); | Строковая функция. Принимает в качестве параметра строку и возвращает ее длину |
| number ToNumber(string a); | Строковая функция. Принимает в качестве параметру строку, состоящую из цифр (например: “12031”) и возвращает соответствующее целочисленное значение |

* 1. **Ввод и вывод данных**

Ввод данных осуществляется с помощью ключевого слова get. В качестве аргумента принимается идентификатор, который необходимо проинициализировать:

get <идентификатор>;

Вывод данных осуществляется с помощью ключевого слова print. В качестве аргумента принимается идентификатор, литерал, или выражение:

print <идентификатор>;

print <литерал>;

print <выражение>;

* 1. **Точка входа**

В языке DDA-2020 точкой входа является функция main. Точка входа не может отсутствовать, также не может быть переопределена. В программе может быть только одна точка входа.

* 1. **Препроцессор**

В языке DDA-2020 препроцессоры не предусмотрены.

* 1. **Соглашения о вызовах**

При генерации кода используется соглашение \_\_stdcall, в котором все параметры передаются в стек справа налево. Освобождением памяти занимается вызываемая подпрограмма, которая очищает стек.

* 1. **Объектный код**

Исходный код языка транслируется в язык ассемблера, а затем в объектный код.

**1.23 Классификация сообщений транслятора**

Транслятор генерирует сообщения об ошибках пользователю. В зависимости от этапа генерации ошибки, сообщение будет иметь специальный префикс.

Классификация ошибок языка DDA-2020 представлена в таблице 1.10.

Таблица 1.10 - Классификация сообщений транслятора

|  |  |
| --- | --- |
| Префикс ошибки | Описание ошибки |
| IN.### | ### - код ошибки. Сообщение, генерируемое на этапе открытия и чтения входного файла. |
| PARM.###: | ### - код ошибки. Сообщение, генерируемое на этапе обработки входных параметров. |
| LEX.###: | ### - код ошибки. Сообщение, генерируемое на этапе лексического анализа. |
| SYN.###: | ### - код ошибки. Сообщение, генерируемое на этапе синтаксического анализа. |
| SEM.###: | ### - код ошибки. Сообщение, генерируемое на этапе семантического анализа. |
| SYST.###: | ### - код ошибки. Сообщение, генерируемое при критической ошибке. |

* 1. **Контрольный пример**

Контрольный пример, написанный на языке DDA-2020, представлен в приложении А.

**Глава 2. Структура транслятора**

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

Исходный код, написанный на языке программирования DDA-2020, является для транслятора входными данными.

Как выходные данные используется ассемблерный код и протоколы работы транслятора, описанные в пункте 2.3.

Компоненты транслятора приведены на рисунке 2.1.



Рис 2.1 - Структура транслятора DDA-2020

Первоначально на вход лексического анализатора передается исходный код. Анализатором проверяется исходный текст на недопустимые символы, выделяет литералы, идентификаторы и ключевые слова, а также формирует таблицы лексем и идентификаторов.

Далее наступает черед синтаксического анализатора, к нему на вход поступает таблица лексем, полученная на этапе лексического анализа. Если программа построена синтаксически правильно, то осуществляется переход к этапу трансляции, стоящему далее, при ином раскладе работа транслятора останавливается.

Генератор кода – этап транслятора, выполняющий генерацию ассемблерного кода на основе полученных данных на предыдущих этапах трансляции. Генератор кода принимает на вход таблицы идентификаторов и лексем и транслирует код на языке DDA-2020, прошедший все предыдущие этапы, в код на языке Ассемблера. Генерация кода реализуется посредством чистой интерпретации, без создания промежуточного представления кода. В финале происходит генерация кода, во время исполнения которого формируется объектный код.

**2.2 Перечень входных параметров транслятора**

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

Таблица 2.1 - Входные параметры транслятора

|  |  |  |
| --- | --- | --- |
| Параметр | Назначение | Тип |
| -in: | Указывает на файл с исходным кодом. Исходный код содержится в файле с расширением \*.txt | Обязательный |
| -out: | Указывает имя выходного файла. Если не указан явно, то имя протокола формируется конкатенацией имени файла исходного кода и постфикса «.asm» | Не обязательный |
| -log: | Указывает имя протокола. Если не указан явно, то имя протокола формируется конкатенацией имени файла исходного кода и постфикса «.log» | Не обязательный |
| -debug | Задает работу транслятора в отладочном режиме. Если указан явно, то информация о работе транслятора дополнительно выводится в консоль разработчика. | Не обязательный |

**2.3 Протоколы, формируемые транслятором**

По итогам своей работы транслятор формирует протокол, согласно заданным входным параметрам. -log: <путь к файлу> - в этом файле находятся информация о входных параметрах, количестве символов исходного кода, таблицы лексем, идентификаторов, работы синтаксического анализатора, дерево разбора.

# Глава 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 производит предварительный разбор текста, преобразующий единый массив текстовых символов в массив токенов. Выходными данными являются таблица лексем и таблица идентификаторов.

Структура лексического анализатора представлена на рисунке 3.1.



Рис 3.1 - Структура лексического анализатора

**3.2 Контроль входных символов**

Таблица допустимых входных символов представлена на рисунке 3.2.

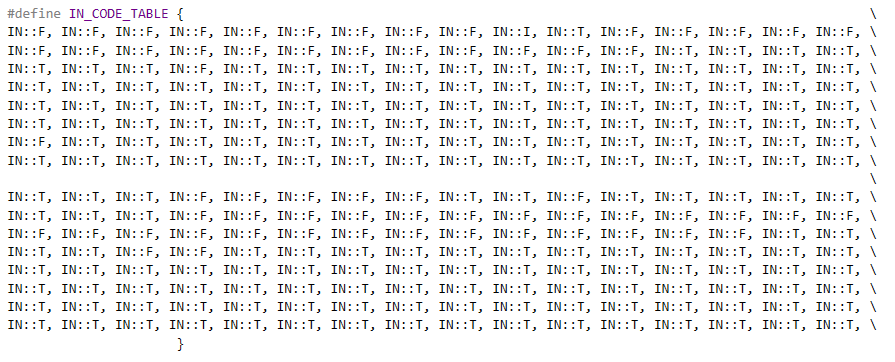


Рисунок 3.2 - Таблица допустимости входных символов

Таблица допустимости была сформирована на основе кодировки Windows-1251. Таблица необходима для проверки входных символов на допустимость. Символы могут быть разрешенными, запрещенными и игнорируемыми. Символы представлены в шестнадцатеричной системе счисления. В таблице записаны различные числовые значения, соответствующие символам в данной таблице:

* «T» - разрешенные алфавитом символы;
* «F» - запрещенные алфавитом символы;
* «I» - символы, которые игнорируются.

**3.3 Удаление избыточных символов**

Избыточными символами являются пробелы, символы табуляции, символы перехода на новую строку. Удаляются в процессе считывания файла.

Описание алгоритма удаления избыточных символов:

1.Посимвольно считываем файл с исходным кодом программы;

2.Если встречаем избыточный символ, то проверяем его необходимость (например, является сепаратором лексем или часть строкового литерала);

3. Избыточные символы игнорируем.

## **3.4 Перечень ключевых слов**

Перечень ключевых слов, сепараторов, символов операций соответствующих им лексем представлен в таблице 3.1.

Таблица 3.1 - Перечень ключевых слов

|  |  |
| --- | --- |
| Цепочка | Лексема |
| number | t |
| string | t |
| bool | t |
| ubyte | t |
| def | d |
| main | m |
| function | f |
| return | r |
| print | p |
| get | g |
| if | ? |
| else | e |
| elif | o |
| for | & |
| in | j |
| step | s |
| and, or | z |
| +, - , \* , = | v |
| >=, <=, ==, != | c |
| &, | | b |
| ~, ++, -- | u |
| .. | . |
| => | > |
| = | = |
| ( | ( |
| ) | ) |
| [ | [ |
| ] | ] |
| { | { |
| } | } |
| , | , |
| ; | ; |
| идентификатор | i |
| литерал | l |

Каждому выражению соответствует детерминированный конечный автомат, по которому происходит разбор данного выражения. На каждый автомат в массиве подаётся токен и с помощью регулярного выражения, соответствующего данному графу переходов, происходит разбор. В случае успешного разбора выражения оно записывается в таблицу лексем. Если выражение является идентификатором или литералом, информация также заносится в таблицу идентификаторов. Структура конечного автомата и пример графа перехода конечного автомата изображены на рисунках 3.3 и 3.4 соответственно.

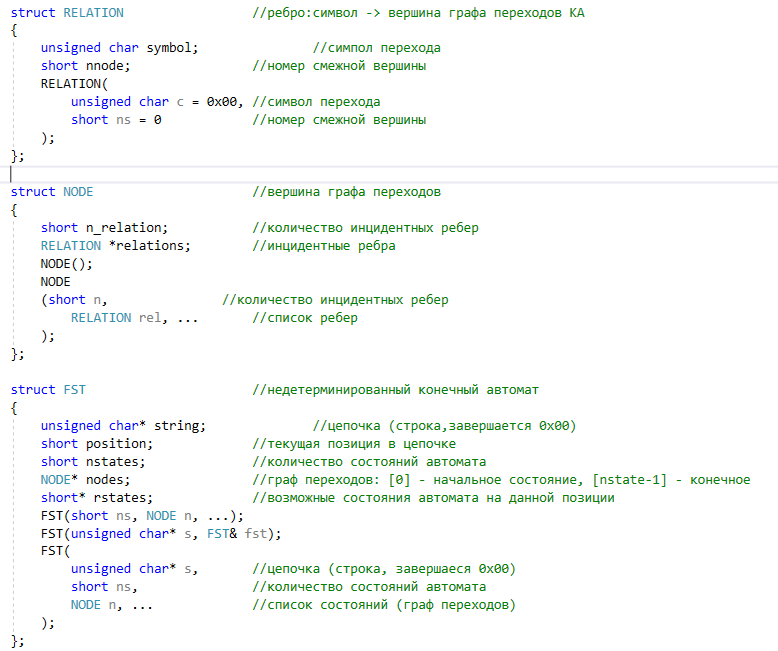


Рисунок 3.3 – Структура конечного автомата

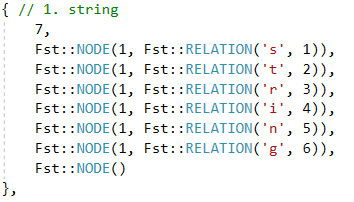
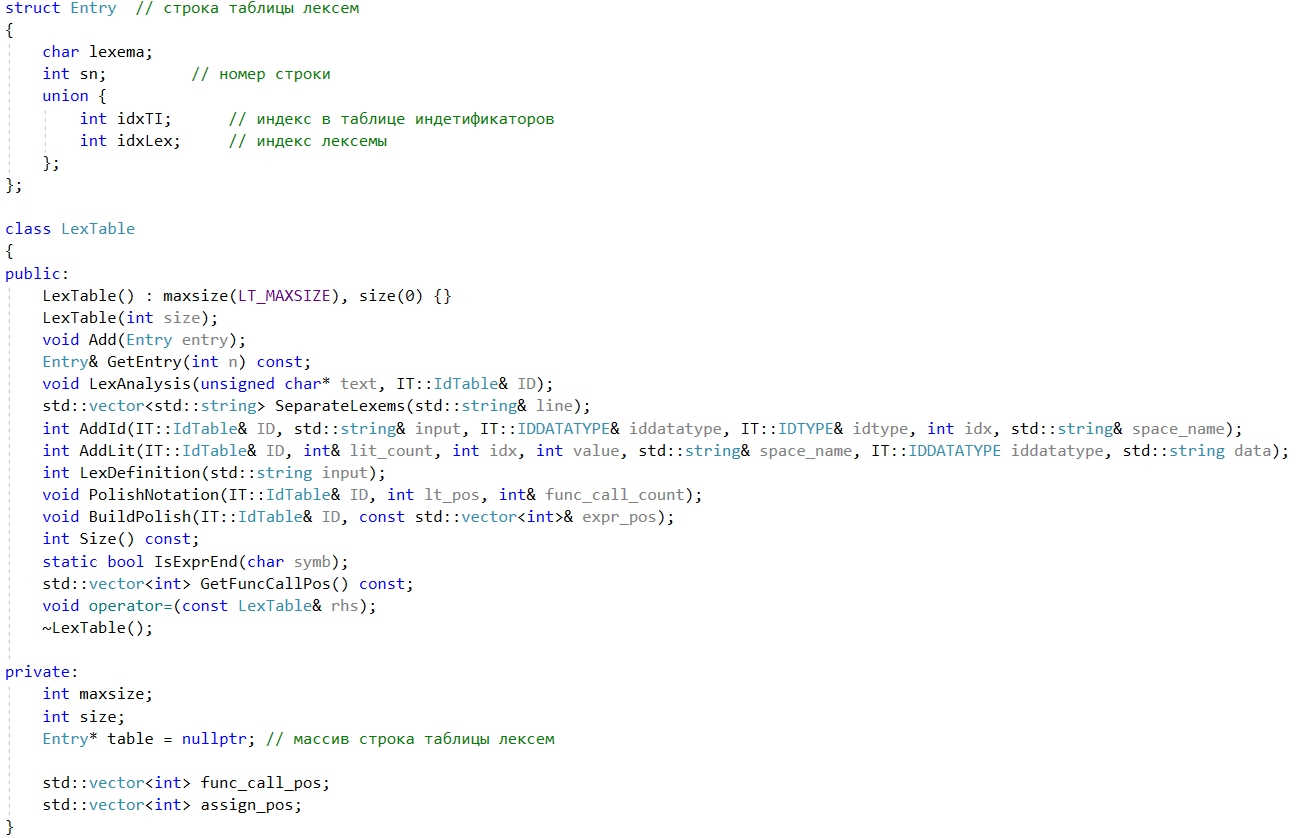


Рисунок 3.4 – Пример реализации графа для токена “string”

## **3.5 Основные структуры данных**

Основными структурами данных являются таблица идентификаторов и таблица лексем. Таблицы заполняются в процессе обработки входного файла лексическим анализатором. Код реализации на языке C++ таблицы лексем представлен на рисунке 3.2. Код реализации таблицы идентификаторов представлен на рисунке 3.3.



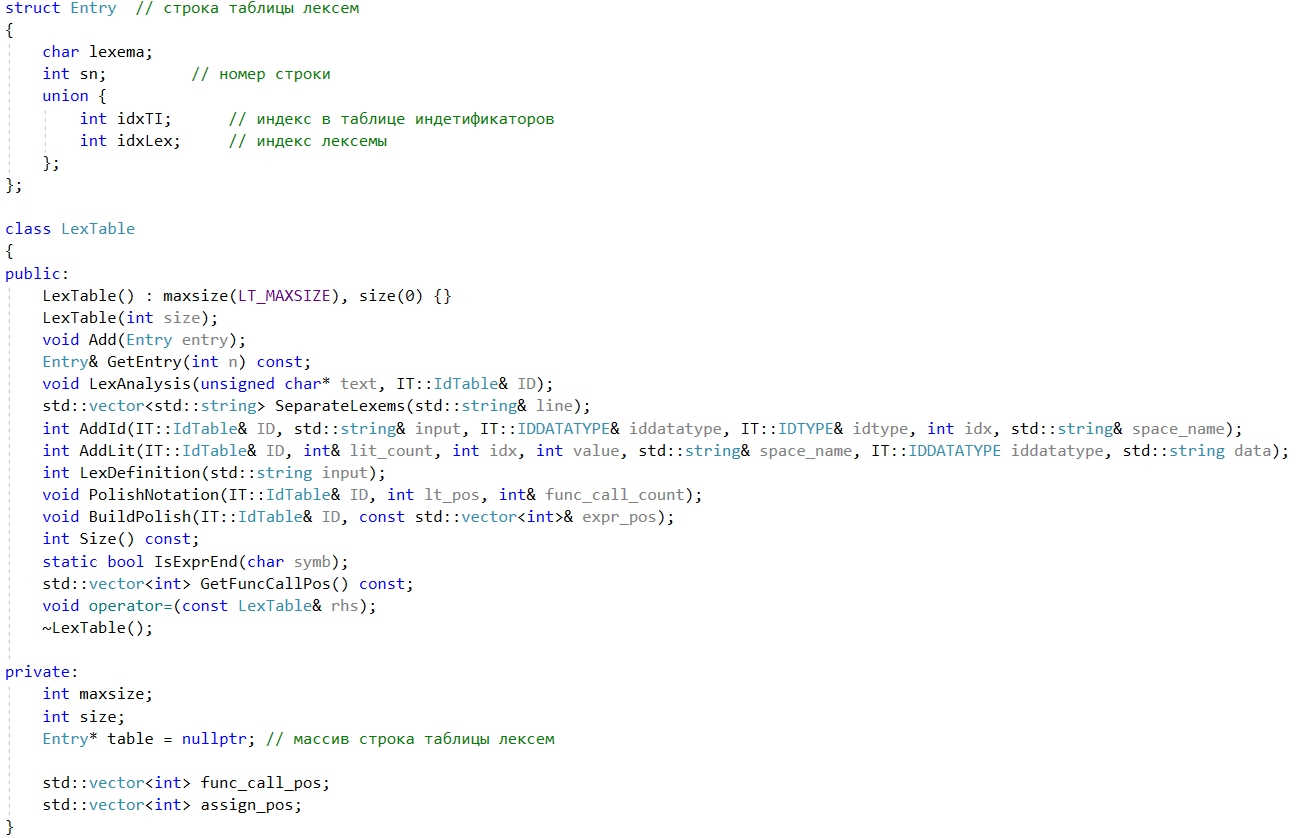
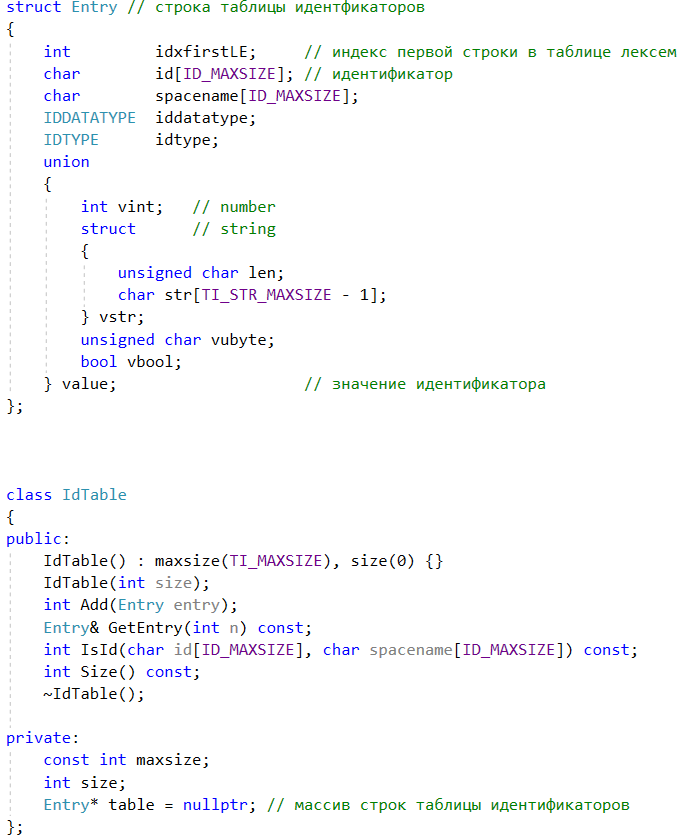


Рис 3.2 - Структура таблицы лексем



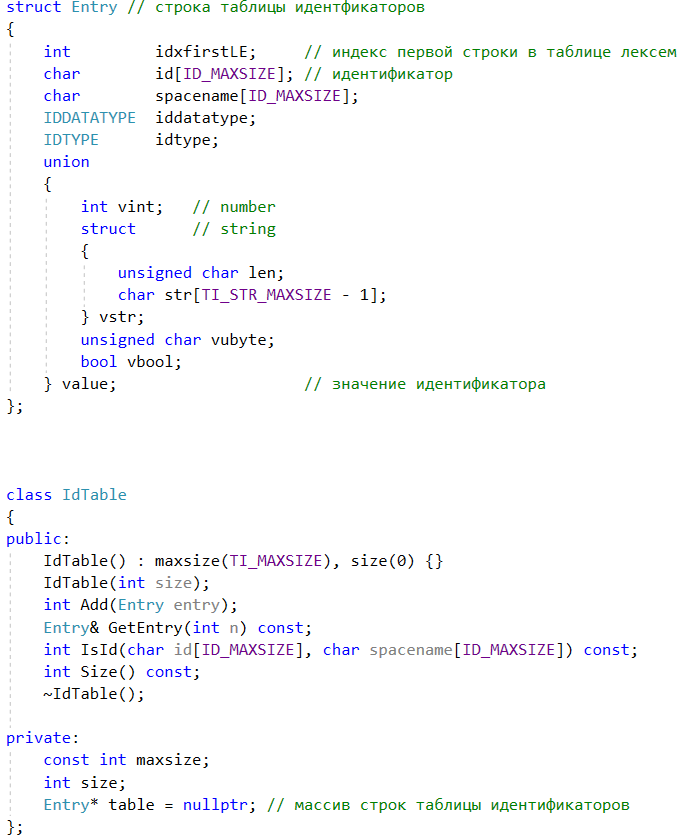


Рис 3.3 - Структура таблицы идентификаторов

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

Префикс сообщений ошибок, генерируемых во время лексического анализа, “LEX.”. Перечень сообщений, генерируемых на этапе лексического анализа, представлен в таблице 3.2.

Таблица 3.2 - Сообщения лексического анализатора

|  |  |
| --- | --- |
| Код | Сообщение |
| 120 | Элемент не распознан |
| 121 | Ошибка при создании лексической таблицы (превышен максимальный размер) |
| 122 | Ошибка при добавлении строки в лексическую таблицу (превышен максимальный размер) |
| 123 | Ошибка при получении строки лексической таблицы (нет элемента) |
| 124 | Ошибка при создании таблицы идентификаторов (превышен максимальный размер) |
| 125 | Ошибка при добавлении в таблицу идентификаторов (превышен максимальный размер) |
| 126 | Ошибка при получении строки таблицы идентификаторов (нет элемента) |
| 127 | Превышен максимальный размер строки |

## **3.7 Принцип обработки ошибок**

В случае обнаружения ошибки, информация об ошибке сохраняется и в конце лексического анализа транслятор прекращает свою работу и записывает информацию об ошибках.

## **3.8 Параметры лексического анализатора и режим его работы**

Текст кода на языке DDA-2020 подается на вход. Если задан режим “-debug”, то результат работы лексического анализатора дополнительно выводится в консоль разработчика.

## **3.9 Алгоритм лексического анализа**

Алгоритм работы лексического анализатора:

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

На рисунке 3.3 продемонстрирована функция, реализованная на языке C++, которая распознает входную лексему на основе конечных автоматов.

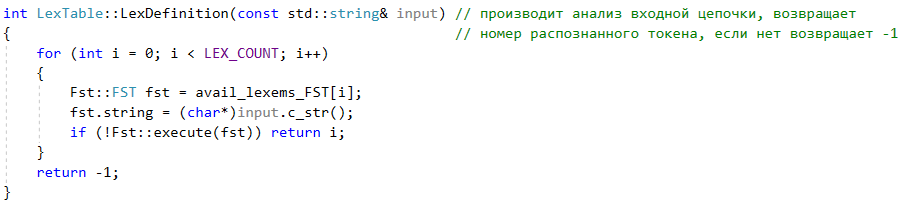


Рис 3.3 – Функция распознавания входной лексемы

## **3.10 Контрольный пример**

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

# Глава 4. Разработка синтаксического анализатора

## **4.1 Структура синтаксического анализатора.**

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

Структура синтаксического анализатора представлена на рисунке 4.1.

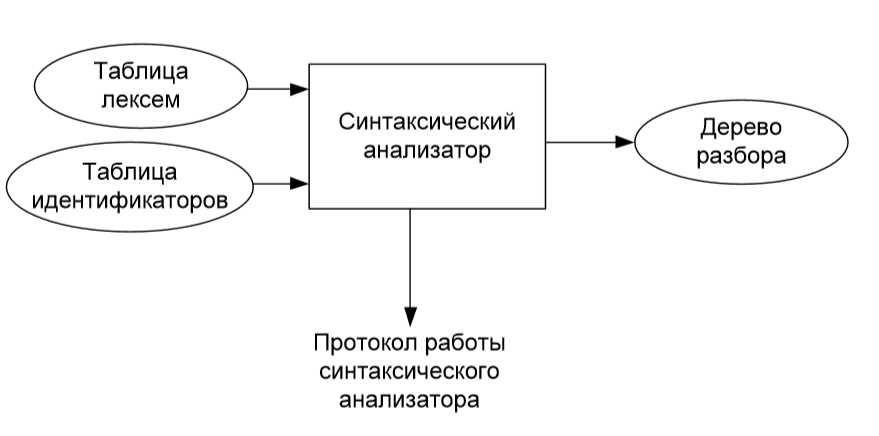


Рисунок 4.1 - Структура синтаксического анализатора

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

Грамматика, описывающая язык DDA-2020 представлена в таблице 4.1.

Таблица 4.1 - грамматика языка DDA-2020

|  |  |
| --- | --- |
| Нетереминалы | Описание |
| S | Правила, описывающие общую структуру программы |
| N | Порождает правила, описывающие инструкции языка |
| E | Порождает правила, описывающие выражения |
| M | Порождает правила, описывающие операции языка |
| P | Порождает правила, описывающие блок параметров формальной функции |
| K | Порождает правила, описывающие блок параметров фактически вызываемой функции |
| F | Порождает правила, описывающие формальные параметры функции |
| W | Порождает правила, описывающие фактические параметры функции |
| C | Порождает правила, описывающие условные выражения |
| I | Порождает правила, описывающие конструкцию условного оператора |
| O | Порождает правила, описывающие тело оператора условия |
| Y | Порождает правила, описывающие инструкции тело оператора условия |
| D | Порождает правила, описывающие индекс элемента |
| R | Порождает правила, описывающие аргументы диапазона цикла |
| Q | Порождает правила, описывающие шаг цикла |
| B | Порождает правила, описывающие тело цикла |
| G | Порождает правила, следующие за телом цикла или телом условия |

## **4.3 Построение конченого магазинного автомата**

Принцип действия конечного магазинного автомата представлен на рисунке 4.2.



Рис. 4.2 - МП-автомат

Формальное описание МП-автомата:



 - множество состояний;

 - алфавит входных символов;

 - специальный алфавит магазинных символов;

-функция переходов автомата , где  - множество подмножеств ;

 - начальное состояние автомата;

- начальное состояние магазина (маркер дна);

- множество конечных состояний.

Конфигурация (текущее состояние автомата) описывается тройкой , где - текущее состояние автомата, - остаток цепочки,  - цепочка-содержимое магазина.

Начальное состояние , - начальное состояние автомата,  - входная цепочка, - маркер дна магазина.

Цепочка  является допустимой (распознается) автоматом , если  и .

Работа автомата 

1. состояние автомата 
2. читает символ  находящийся под головкой (сдвигает ленту);
3. не читает ничего (читает , не сдвигает ленту);
4. из  определяет новое состояние , если  или .
5. читает верхний (в стеке) символ  и записывает цепочку  т.к. , при этом, если , то верхний символ магазина просто удаляется.

Работа автомата заканчивается .

## **4.4 Основные структуры данных**

В приложении Г представлены основные структуры данных и правила перехода, используемые на этапе синтаксического анализа.

## **4.5 Описание алгоритма синтаксического разбора**

Входные символы и лексемы в форме Грейбах находятся в ленте на входе конечного автомата.

1) Если лента не пустая, переходим далее следующему пункту, иначе переходим к пункту 5.

2) Если на верхушке магазина нетерминальный символ.

2.1) Если есть такое правило, то переходим к следующему пункту.

2.1.1) Если цепочка есть, возвращаем NS\_OK. Переходим к пункту 4.

2.1.2) Иначе восстанавливаем состояние. Переходим к пункту 4.

2.2) Иначе возвращаем ошибку. Переход к пункту 4.

3) Если на верхушке терминал и он совпадает с символом на ленте, то удаляем его из стека и продвигаем ленту. Переход к пункту 4.

4) Повторяем шаг, переходим к пункту 1.

5) Конец работы.

## **4.6 Структура и перечень сообщений синтаксического анализатора**

Префикс сообщений “SYN.”. Перечень сообщений, генерируемых на этапе синтаксического анализа, представлен в таблице 4.3.

Таблица 4.3 - Перечень сообщений

|  |  |
| --- | --- |
| Код | Сообщение |
| 600 | Неверная структура программы |
| 601 | Ошибочный оператор |
| 602 | Ошибка в выражении |
| 603 | Ошибка в параметрах функции |
| 604 | Ошибка в параметрах вызываемой функции |
| 605 | Ошибка в арифметическом выражении |
| 606 | Ошибка в условном операторе |
| 607 | Ошибка в условной конструкции |
| 608 | Ошибка в операторах тела условной конструкции |
| 609 | Ошибка в индексе |
| 610 | Ошибка в аргументах диапазона цикла |
| 611 | Ошибка в шаге цикла |
| 612 | Ошибка в теле цикла |
| 613 | Ошибка в операторах после тела цикла или условия |

## **4.7 Параметры синтаксического анализатора и режимы его работы**

Таблицы идентификаторов и лексем являются входными параметрами для синтаксического анализатора. Эти таблицы мы получаем в ходе лексического анализа. В конце после разбора формируется дерево разбора, которое выводится в протокол работы –log, а также в консоль, если программы была запущена в режиме “debug” (ключ запуска -debug).

## **4.8 Принцип обработки ошибок**

При обнаружении ошибки в цепочке какого-либо правила, синтаксического анализатора идет вверх по дереву разбора, пока не найдет верный вариант. Иначе запоминается самая глубокая ошибка, которая выводится в протокол работы.

## **4.9 Контрольный пример**

Результатом работы синтаксического анализатора является дерево разбора. Результат работы синтаксического анализатора представлен в приложении В.

# Глава 5. Разработка семантического анализатора

1. 1. **Структура семантического анализатора**

Семантический анализатор состоит из набора функций для проверки правильности исходной программы. Функции анализатора выполняются на различных этапах работы транслятора. Структура семантического анализатора представлена на рисунке 5.1.

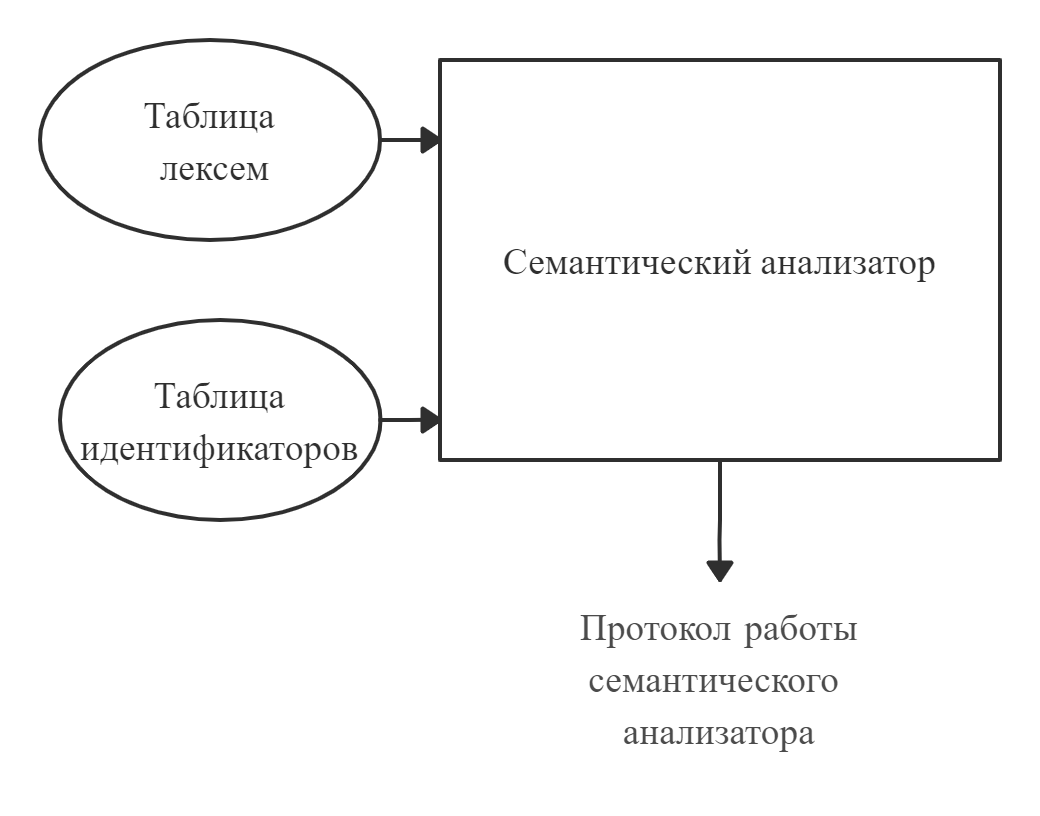


Рисунок 5.1 - Структура семантического анализатора

* 1. **Функции семантического анализатора**

В таблице 5.1 перечислены семантические проверки, осуществляемые языком DDA-2020. Некоторые проверки встроены непосредственно в код этапов транслятора.

Таблица 5.1 - Семантические проверки

|  |  |
| --- | --- |
| Фаза выполнения | Семантическая проверка |
| Лексический анализ | Наличие уникальной функции main |
| Лексический анализ | Усечение слишком длинных идентификаторов до 5 символов |
| Лексический анализ | Сначала осуществляется проверка на ключевые слова, а затем на идентификатор. Не допускаются идентификаторы, совпадающие с ключевыми словами |
| Лексический анализ | Уникальность именования идентификатора, в пределах области видимости |
| Лексический анализ | Предварительное объявление, применяемых идентификаторов и функций |
| Лексический анализ | Усечение слишком длинного значения строкового литерала |
| Отдельная фаза | Соответствие типов формальных и фактических параметров при вызове функций |
| Отдельная фаза | Типы переменных в выражениях должны быть согласованы между собой |
| Отдельная фаза | Все операнды в выражениях и операциях должны иметь типы, допустимые для данного выражения или операции |
| Отдельная фаза | Возвращаемый тип функции должен соответствовать типу функции |
| Отдельная фаза | В индексе используется целочисленный тип |
| Отдельная фаза | Переменная цикла и диапазон цикла имеют целочисленный тип |

Функции, реализующие проверку правил, представлены в таблице 5.2.

Таблица 5.2 - Функции реализующие семантические проверки

|  |  |
| --- | --- |
| Функция | Описание |
| CheckParameters | Проверка, на соответствие типов формальных и фактических параметров функций |
| ChechkExprType | Проверка на допустимость типов в выражениях, операциях, условном операторе и цикле |

* 1. **Структура и перечень сообщений семантического анализатора**

Префикс сообщений “SEM.”, генерируемых на этапе семантического анализа.

Сообщения, генерируемые при выполнении семантических проверок, представлены в таблице 5.3.

Таблица 5.3 - Перечень сообщений

|  |  |
| --- | --- |
| Код | Сообщение |
| 200 | Функция main должна быть уникальна |
| 201 | Отсутствует функция main |
| 202 | Использование необъявленного идентификатора |
| 203 | Возвращаемый тип не соответствует типу функции |
| 204 | Идентификатор с таким именем уже задан в этой области видимости |
| 205 | Недопустимо объявление переменной в глобальной области |
| 206 | Имя функции не может совпадать с именем функции стандартной библиотеки |
| 207 | Неверное количество параметров у вызываемой функции |
| 208 | Неверные типы у вызываемой функции |
| 209 | Недопустимо использование разных типов данных в одном выражении |
| 210 | Тип присваемого значения не соответствует типу идентификатора |
| 211 | Сравнение разных типов данных |
| 212 | Недопустимо использование арифметических операций над типом string |
| 213 | Индексация допустима только к типу string |
| 214 | Индекс должен иметь целочисленный тип |
| 215 | Переменная цикла должна иметь целочисленный тип |
| 216 | Диапазон цикла должен иметь целочисленный тип |

* 1. **Принцип обработки ошибок**

Семантические ошибки, генерируемые в одной фазе обработки накапливаются. В конце работы фазы транслятор прекращает свою работу и в протокол работы транслятора выводится соответствующее сообщения об сохраненных ошибках.

* 1. **Контрольный пример**

Результат работы семантических проверок описан в главе 8.

**Глава 6. Преобразование выражений**

**6.1 Выражения, допускаемые языком**

В языке DDA-2020 допускаются выражения с использованием числовых идентификаторов, литералов и вызовов функций. Также предусмотрены следующие арифметические операции:

* сложения «+»;
* вычитания «-»;
* умножения «\*»;
* деления «/»;
* инкремент «++»;
* декремент «--».

Битовые операции:

* и «&»;
* или «|»;
* не «~».

Есть возможность изменять приоритет выполнения арифметических операций при помощи скобок:

* «(»;
* «)».

Из-за возможности использования в выражениях вызова функции, также используются следующие операции:

* «,» (разделение параметров вызываемой функции);
* «[» (начало вызова функции);
* «]» (конец вызова функции).

За счет заключения операции в скобки происходит повышение его приоритета, а значит при вычислении всего выражения операция с более высоким приоритетом будет вычисляться раньше операции с меньшим приоритетом.

Приоритетность операций представлена в таблице 6.1.

Таблица 6.1 - Приоритетность операций

|  |  |
| --- | --- |
| Операция | Приоритет |
| «(», «)» | 0 |
| «,» | 1 |
| «+», «-» | 2 |
| «\*», «/», «&»,«|», «~» | 3 |
| «[», «]»,«++», «--» | 4 |

**6.2 Польская запись и принцип ее построения**

Польская запись -форма записи математических и логических выражений, в которой операнды расположены перед знаками операций.

Выражение читается слева направо. Каждая операция выполняется над двумя операндами, непосредственно стоящими перед знаком этой операции. Последовательность операндов и знак операции в выражении заменяется результатом этой операции. Результатом вычисления всего выражения становится результат последней вычисленной операции. Приоритетность операций приведена в таблице 6.1.

Алгоритм построения:

* исходная строка: выражение;
* результирующая строка: польская запись;
* стек: пустой;
* исходная строка просматривается слева направо;
* операнды переносятся в результирующую строку в порядке их следования;
* операция записывается в стек, если стек пуст или в вершине стека лежит отрывающая скобка;
* операция выталкивает все операции с большим или равным приоритетом в результирующую строку;
* запятая не помещается в стек, и если в стеке есть операции, то все выбираются в строку;
* открывающая скобка помещается в стек;
* закрывающая скобка выталкивает все операции до открывающей скобки, после чего обе скобки уничтожаются;
* квадратная открывающая скобка помещается в стек;
* квадратная закрывающая скобка выталкивает все до открывающей квадратной скобки и генерирует последовательность @
* по концу разбора исходной строки все операции, оставшиеся в стеке, выталкиваются в результирующую строку.

**6.3 Программная реализация обработки выражений**

Преобразование выражений в обратный польский формат происходит перед этапом генерации кода, после прохождения всех семантических и синтаксических проверок. Программная реализация алгоритма преобразования выражений к польской записи представлена в приложении Д.

**6.4 Контрольный пример**

Контрольный пример разбора выражения содержится в таблице 6.2.

Таблица 6.2 - Разбор выражения

|  |  |  |
| --- | --- | --- |
| Выражение | Стек | Результирующая строка |
| (a + b) \* f(x,y) |  |  |
| a + b) | ( |  |
| + b) \* f(x,y) | ( | a |
| b) \* f(x,y) | ( + | b |
| ) \* f(x,y) | ( + | a b |
| \* f(x,y) |  | a b + |
| f(x,y) | \* | a b + |
| x,y) | \* [ | a b + |
| ,y) | \* [ | a b + x |
| y) | \* [ | a b + x |
| ) | \* [ | a b + x y |
|  | \* | a b + x y @ |
|  |  | a b + x y @ \* |

# Глава 7. Генерация кода

## **7.1 Структура генератора кода**

Заключительным этапом трансляции языка DDA-2020 является генерация кода. Генератор принимает на вход таблицу лексем, таблицу идентификаторов и дерево разбора. На выходе получается файл с исходным кодом на ассемблере, который будет являться результатом работы транслятора. В случае возникновения ошибок генерация кода не будет осуществляться. Структура генератора кода DDA-2020 представлена на рисунке 7.1.



Рисунок 7.1 – Структура генератора кода

## **7.2 Представление типов данных в памяти**

Элементы таблицы идентификаторов расположены сегментах .data и .const языка ассемблера. Соответствия между типами данных идентификаторов на языке DDA-2020 и на языке ассемблера приведены в таблице 7.1.

Таблица 7.1 – Соответствие типов языков программирования

|  |  |  |
| --- | --- | --- |
| Тип идентификатора на языке DDA-2020 | Тип идентификатора на языке ассемблера | Пояснение |
| number | SDWORD | Хранит целочисленный тип данных |
| string | SBYTE | Хранит указатель на начало строки. Строка должна завешаться нулевым символом |
| ubyte | BYTE | Хранит целочисленный беззнаковый тип данных |
| bool | BYTE | Хранит логический тип данных |

## **7.3 Статическая библиотека**

В языке DDA-2020 предусмотрена статическая библиотека. Статическая библиотека содержит функции, написанные на языке C++ и автоматически подключается на начальном этапе генерации кода.

Функции стандартной библиотеки представлены в таблице 7.2.

Таблица 7.2. - Функции стандартной библиотеки

|  |  |
| --- | --- |
| Функция на языке С++ | Описание |
| void print\_number(int n) | Функции для вывода в стандартный поток значения целочисленного идентификатора/литерала типа number. |
| void print\_string(char\* str) | Функции для вывода в стандартный поток значения строкового идентификатора/литерала. |
| void print\_bool (bool b) | Функции для вывода в стандартный поток значения логического идентификатора/литерала. |
| void print\_ubyte(unsigned char ub) | Функции для вывода в стандартный поток значения целочисленного идентификатора/литерала типа ubyte. |
| void get\_number(int& n); | Функция для считывания целочисленного идентификатора из консоли |
| void get\_string (char\* str); | Функция для считывания строкового идентификатора из консоли |
| void get\_bool (bool& b); | Функция для считывания логического идентификатора из консоли |
| void get\_ubyte(unsigned char& ub); | Функция для считывания целочисленного идентификатора типа ubyte из консоли |
| bool strcopy(char\* str1, char\* str2) | Функция, копирующая значение строки str1 в строку str2 |
| bool strconcat(char\* str1, char\* str2) | Строковая функция. Возвращает результат конкатенации строк str1 и str2, записанный в строку str2 |
| int strlength(char\* str) | Строковая функция. Вычисляет и возвращает длину строки str |
| int ToNumber(char\* str) | Строковая функция. Принимает строку составленную из чисел и возвращает соответствующее значение в целочисленном формате |

## **7.4 Особенности алгоритма генерации кода**

В языке DDA-2020 генерация кода строится на основе таблиц лексем и идентификаторов. Общая схема работы генератора кода представлена на рисунке 7.2.

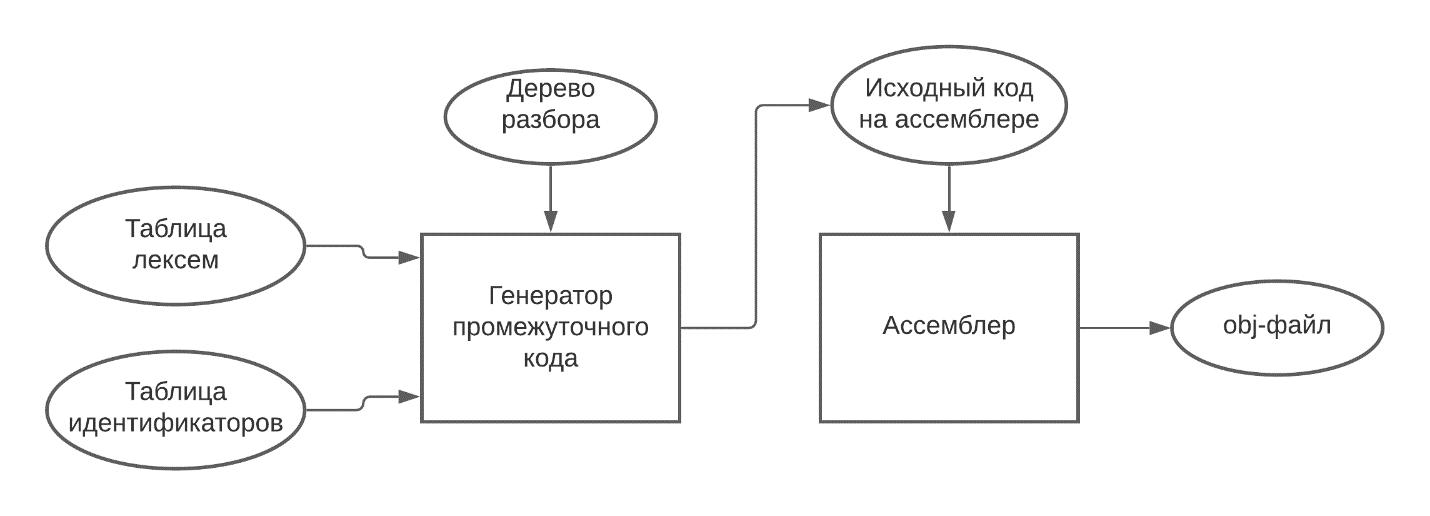


Рисунок 7.2 - Структура генератора кода

## **7.5 Контрольный пример**

Результат генерации ассемблерного кода на основе контрольного примера из приложения А приведен в приложении Е.

# 8. Тестирование транслятора

## **8.1 Общие положения**

При возникновении ошибки на каком-либо этапе трансляции, она обрабатывается в главном файле программы: ошибка выводится на консоль и записывается в протокол работы.

## **8.2 Результаты тестирования**

В таблице 8.1 приведены ошибки, генерируемые в процессе считывания входного файла, а также в процессе лексического, синтаксического и семантического анализов.

Таблица 8.1 – Результаты тестирования транслятора

|  |  |
| --- | --- |
| Исходный код | Диагностическое сообщение |
| maўn  {  } | IN.111: Недопустимый символ в исходной файле (-in), строка 1, позиция 3 |
| main  {  print 'кавычки";  } | LEX.120: Элемент не распознан, строка 3, позиция 2 |
| {  return 1;  } | SEM.201: Отсутствует функция main |
| main  {  def number n = “123”;  return 1;  } | SEM.210: Тип присваемого значения не соответствует типу идентификатора, строка 3 |
| number function f (string n)  {  if(n > '1') => n = n \* 5;  return true;  }  main  {  def ubyte res = f(123);  return 1;  } | SEM.208: Неверные типы параметров у вызываемой функции, строка 8  SEM.211: Сравнение разных типов данных, строка 3  SEM.212: Недопустимо использование арифметических операций над типом string, строка 3  SEM.209: Недопустимо использование разных типов данных в одном выражении, строка 3  SEM.203: Возвращаемый тип не соответствует типу функции, строка 4 |
| main  {  def number n = (3+(2 \* 5);  } | SYN.605: Ошибка в арифметическом выражении, строка 3 |
| main  {  def number i;  for i in 0.. in i++  { }  } | SYN.610: Ошибка в аргументах диапазона цикла, строка 4 |
| main  {  if(1==2) => return 2;  else => return 1;  else => return 0;  } | SYN.607: Ошибка в условной конструкции, строка 5 |

# Заключение

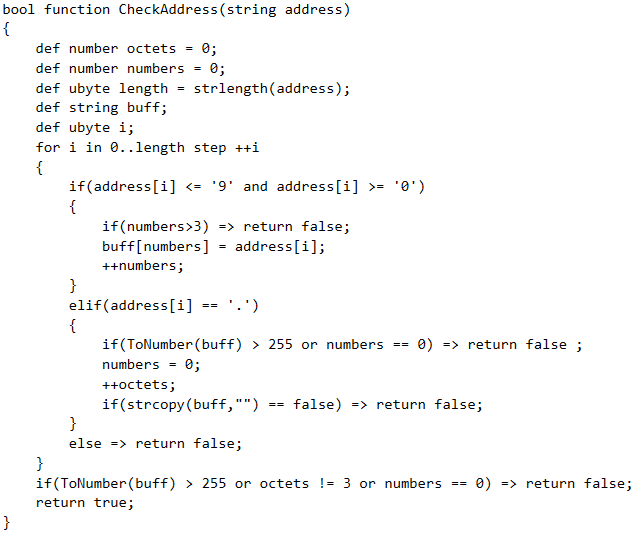
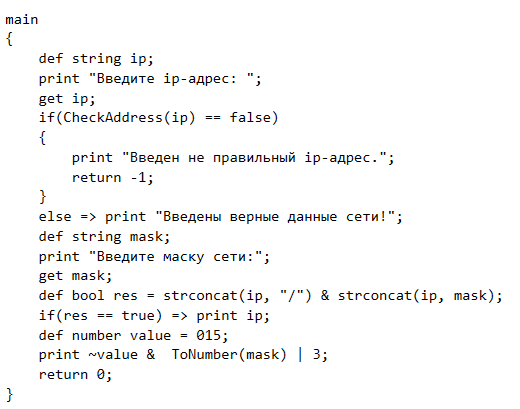
В ходе выполнения курсовой работы был разработан транслятор для языка программирования DDA-2020.

* Реализованы 4 арифметических оператора для вычисления выражений;
* Создано 3 битовых оператора для вычисления выражений;
* Реализованы 6 операторов сравнения;
* Добавлены 2 унарные операции (инкремент, декремент);
* Создано 4 типа данных;
* Поддержка оператора ввода и оператора вывода для всех типов данных;
* Создана стандартная библиотека, содержащая 4 функции;
* Реализована условная конструкция;
* Создан оператор цикла;
* Добавлено возможность проверки нескольких условий в условной конструкции;
* Обрабатывается более 40 возможных исключительных ситуаций;
* Создано 50 различных лексем языка;
* Более 3000 строк кода.

# Список использованных источников

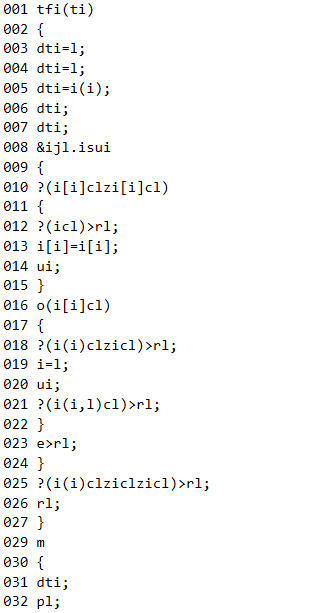
1. Ахо, А. Компиляторы: принципы, технологии и инструменты / А. Ахо, Р. Сети, Дж. Ульман. – M.: Вильямс, 2003. – 768с.
2. Справочник по ассемблеру макросов (Майкрасофт) [Электронный ресурс] – Режим доступа: docs.microsoft.com/ru-ru/cpp/assembler/masm/microsoft-macro-assembler-reference
3. Справочник по языку C++ [Электронный ресурс] – Режим доступа: https://en.cppreference.com/w/
4. Рао, С. Освой самостоятельно C++ за 21 день / Сиддхартха Рао – 7-е изд. Москва: Вильямс, 2013. – 685 с.
5. Страуструп, Б. Принципы и практика использования C++ / Б. Страуструп – 2009 – 1238 с

# Приложение А

Исходный код программы на языке DDA-2020 

# Приложение Б

Таблица лексем контрольного примера



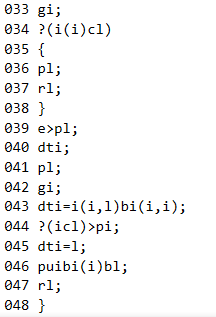
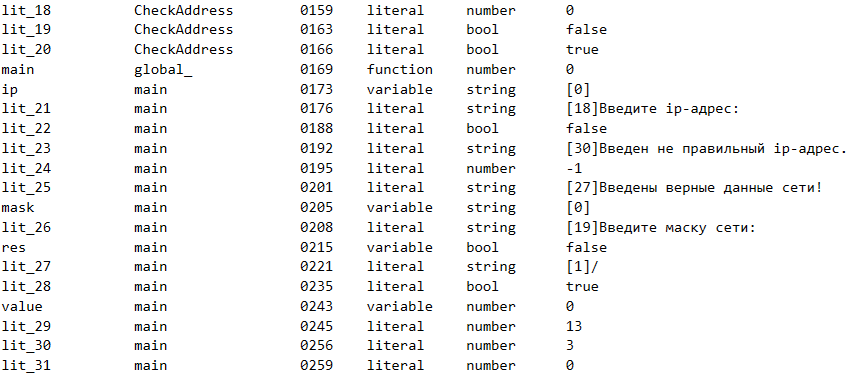
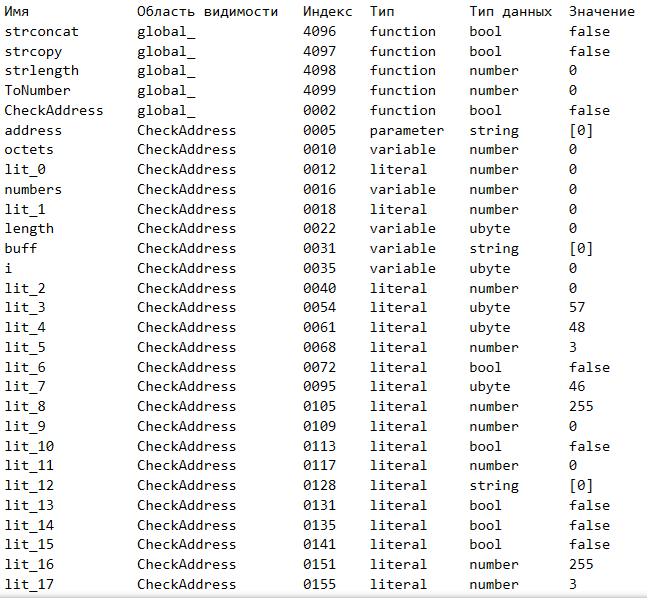
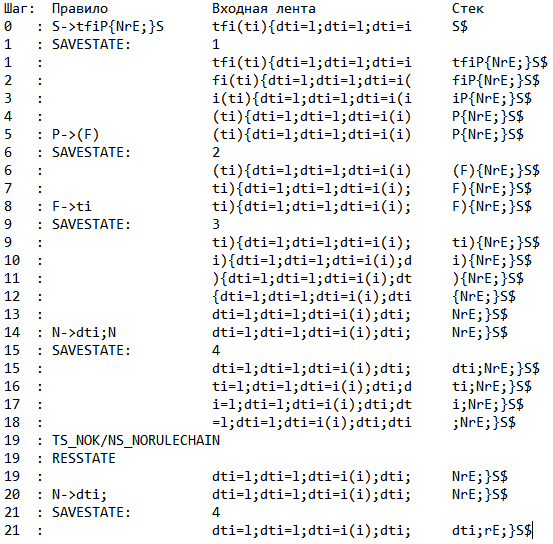


Таблица идентификаторов контрольного примера

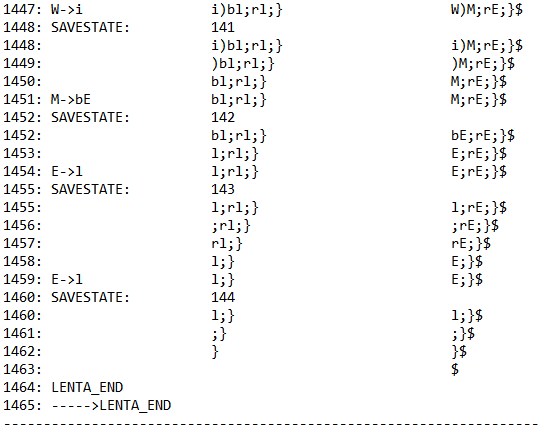


# Приложение В

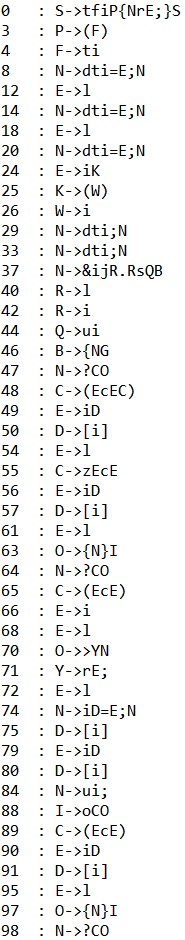
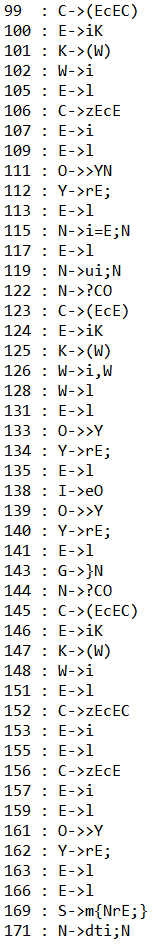
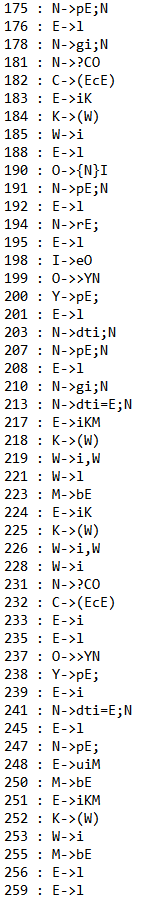
Фрагмент протокола разбора исходного кода синтаксическим анализатором



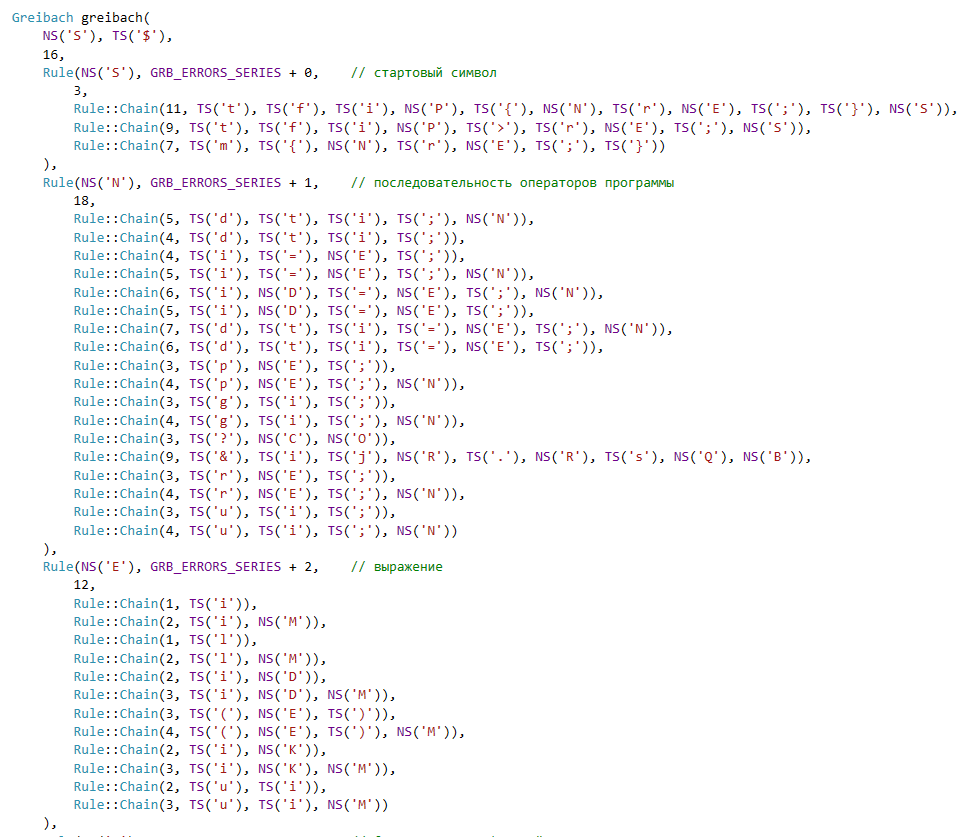
..

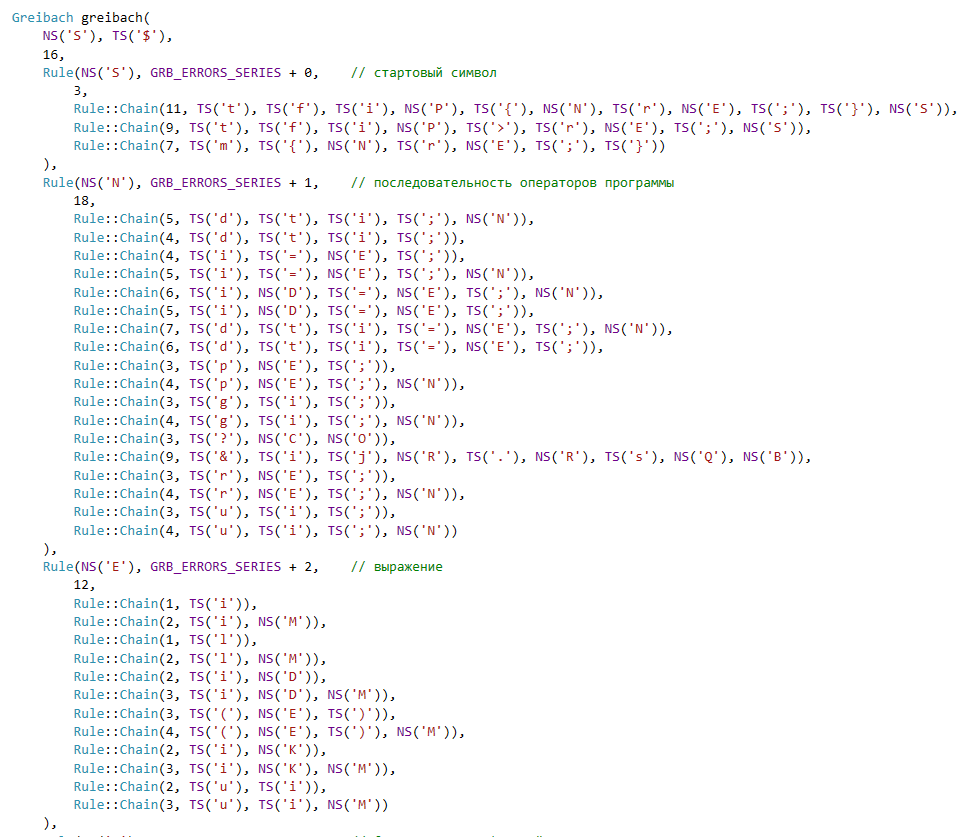


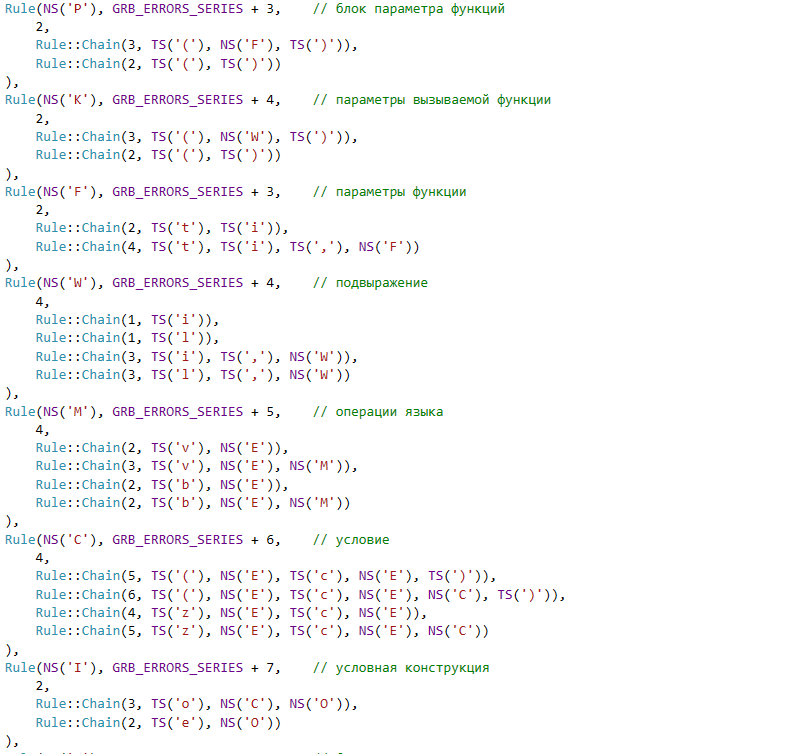
Дерево разбора контрольного примера

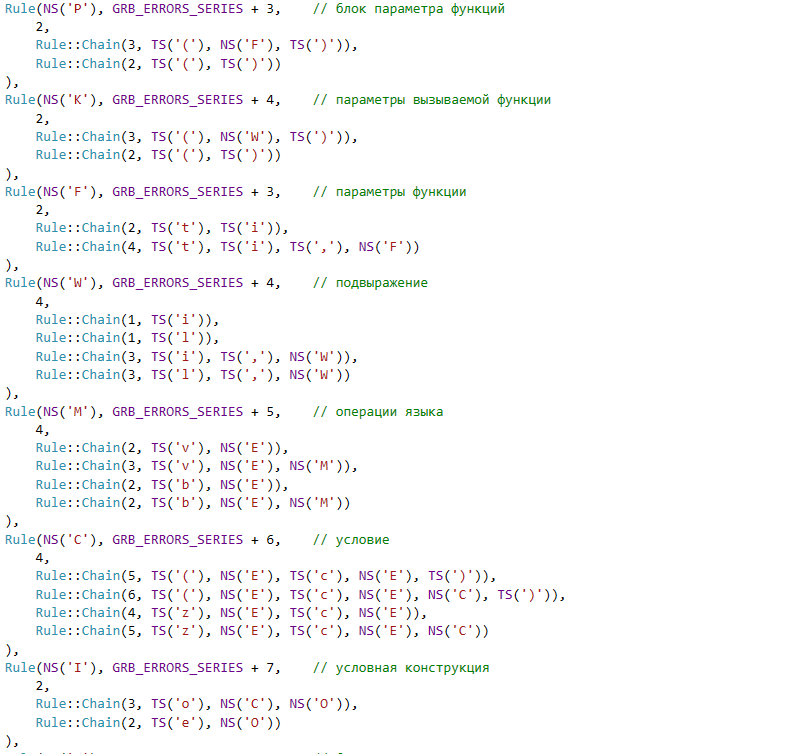
  

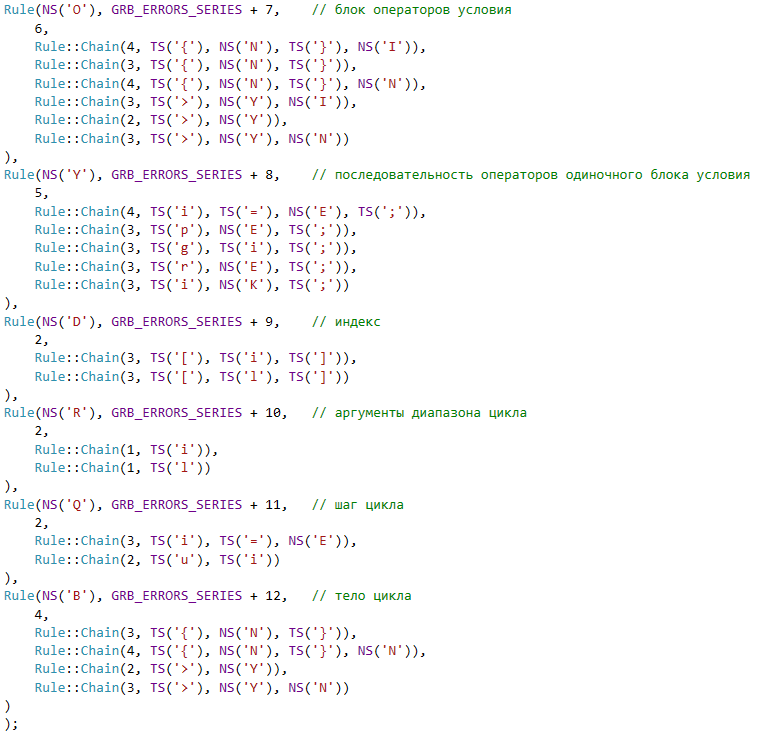
# Приложение Г

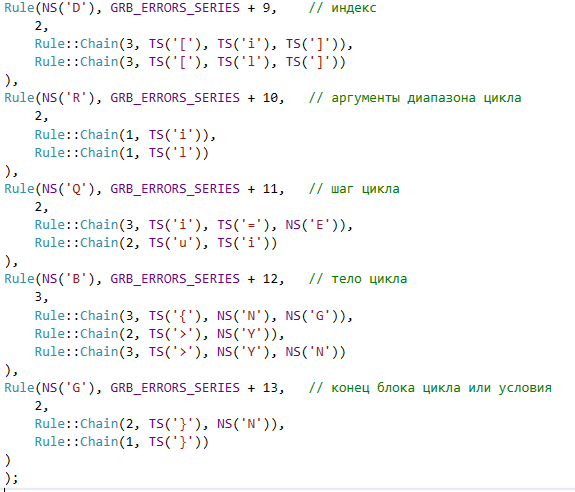
Правила, описывающие грамматику языка





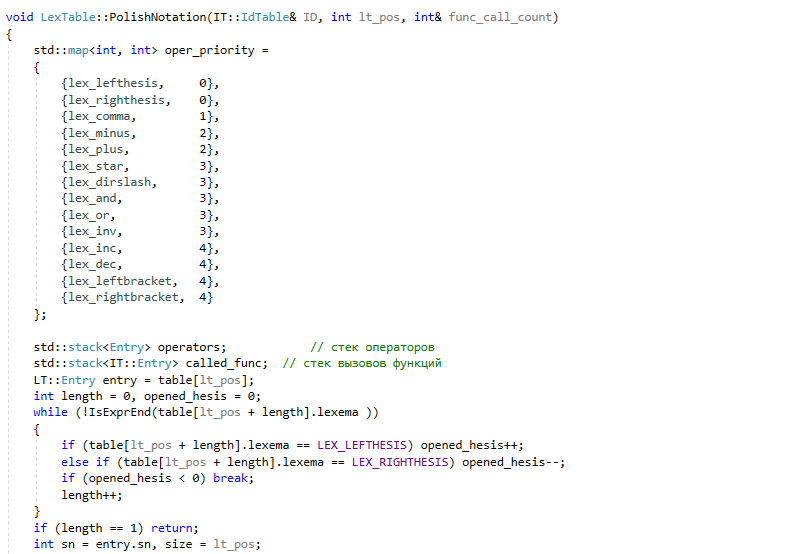
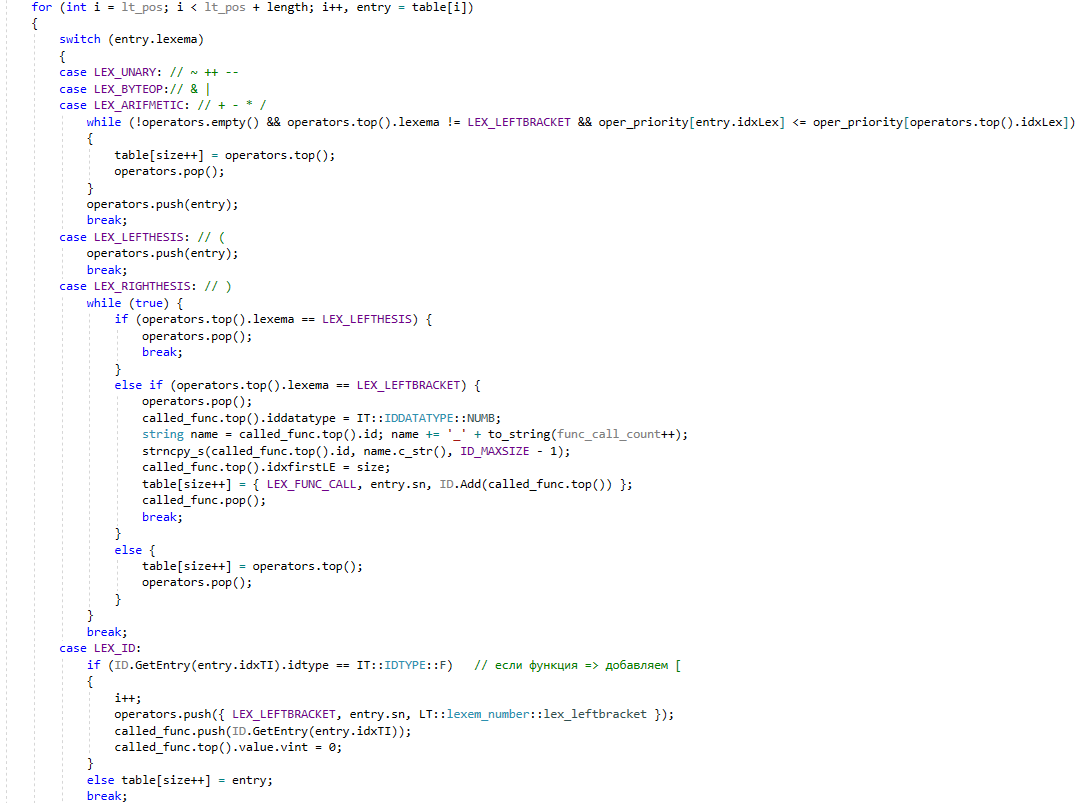
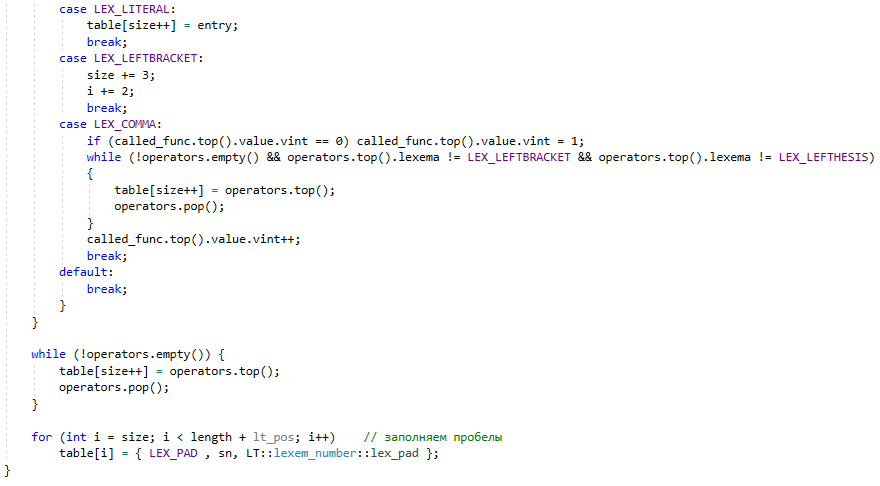
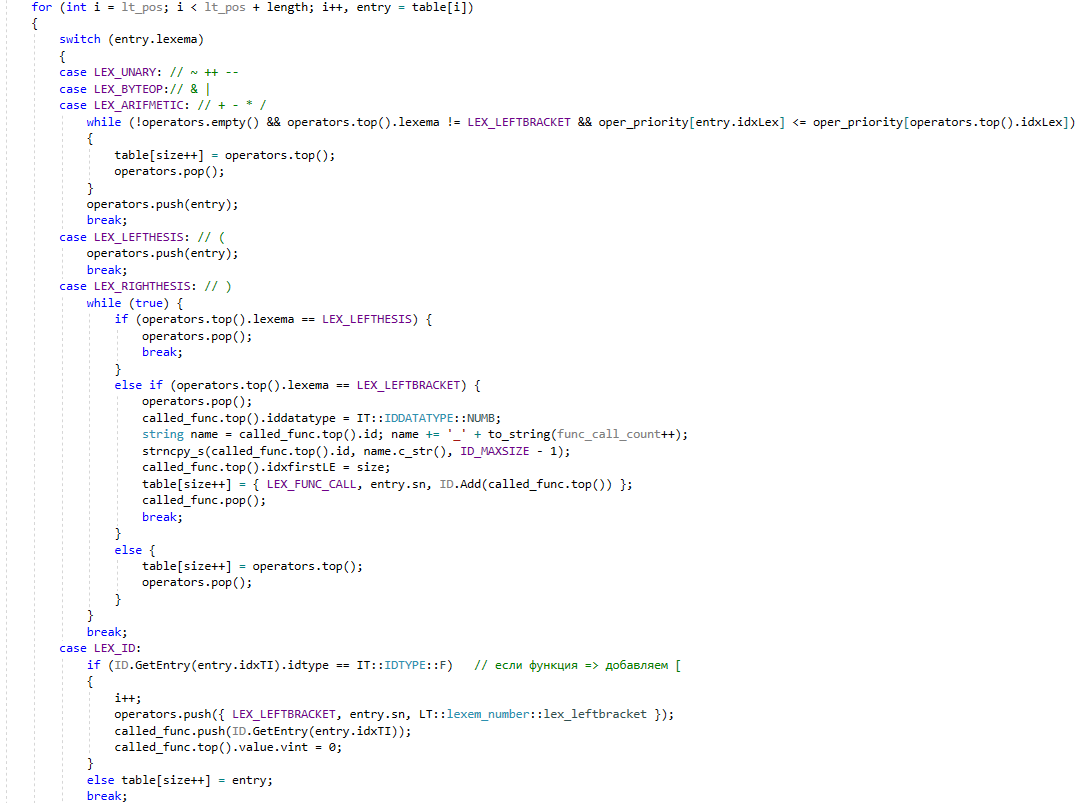






# Приложение Д

Функция преобразования выражения в обратный польский формат

# Приложение Е

Результат генерации контрольного примера в ассемблер