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 ODU-2022»

Выполнил студент Окулич Дмитрий Юрьевич

(Ф.И.О.)

Руководитель проекта ст.п. Наркевич Аделина Сергеевна

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

Заведующий кафедрой к.т.н., доц. Пацей Наталья Владимировна

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

Консультант ст.п. Наркевич Аделина Сергеевна

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

Нормоконтролер ст.п. Наркевич Аделина Сергеевна

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

Курсовой проект защищен с оценкой

Минск 2022

**Содержание**

[**Введение** 5](#_Toc27604552)

[**1. Спецификация языка программирования** 6](#_Toc27604553)

[**1.1.** **Характеристика языка программирования** 6](#_Toc27604554)

[**1.2.** **Определение алфавита языка программирования** 6](#_Toc27604555)

[**1.3.** **Применяемые сепараторы** 6](#_Toc27604556)

[**1.4.** **Применяемые кодировки** 7](#_Toc27604557)

[**1.5.** **Типы данных** 7](#_Toc27604558)

[**1.6.** **Преобразование типов данных** 8](#_Toc27604559)

[**1.7.** **Идентификаторы** 8](#_Toc27604560)

[**1.8.** **Литералы** 9](#_Toc27604561)

[**1.9.** **Объявление данных** 9](#_Toc27604562)

[**1.10.** **Инициализация данных** 9](#_Toc27604563)

[**1.11.** **Инструкции языка** 10](#_Toc27604564)

[**1.12.** **Операции языка** 10](#_Toc27604565)

[**1.13.** **Выражение и их вычисление** 11](#_Toc27604566)

[**1.14.** **Конструкции языка** 11](#_Toc27604567)

[**1.15.** **Область видимости переменных** 12](#_Toc27604568)

[**1.16.** **Семантические проверки** 12](#_Toc27604569)

[**1.17.** **Распределение оперативной памяти на этапе выполнения** 13](#_Toc27604570)

[**1.18.** **Стандартная библиотека и её состав** 13](#_Toc27604571)

[**1.19.** **Ввод и вывод данных** 14](#_Toc27604572)

[**1.20.** **Точка** **входа** 14](#_Toc27604573)

[**1.21.** **Препроцессор** 14](#_Toc27604574)

[**1.22.** **Соглашения о вызовах** 14](#_Toc27604575)

[**1.23.** **Объектный код** 15](#_Toc27604576)

[**1.24.** **Классификация сообщений транслятора** 15](#_Toc27604577)

[**1.25.** **Контрольный пример** 15](#_Toc27604578)

[**2. Структура транслятора** 15](#_Toc27604579)

[**2.1 Компоненты транслятора, их назначение и принципы взаимодействия** 15](#_Toc27604580)

[**2.2 Перечень входных параметров транслятора** 16](#_Toc27604581)

[**2.3 Протоколы, формируемые транслятором** 17](#_Toc27604582)

[**3 Разработка лексического анализатора** 17](#_Toc27604583)

[**3.1 Структура лексического анализатора** 17](#_Toc27604584)

[**3.2 Контроль входных символов** 18](#_Toc27604585)

[**3.3 Удаление избыточных символов** 19](#_Toc27604586)

[**3.4 Перечень ключевых слов** 19](#_Toc27604587)

[**3.5 Основные структуры данных** 21](#_Toc27604588)

[**3.6 Принцип обработки ошибок** 23](#_Toc27604589)

[**3.7 Структура и перечень сообщений лексического анализатора** 23](#_Toc27604590)

[**3.8 Параметры лексического анализатора** 24](#_Toc27604591)

[**3.9 Алгоритм лексического анализа** 24](#_Toc27604592)

[**3.10 Контрольный пример** 24](#_Toc27604594)

[**4. Разработка синтаксического анализатора** 25](#_Toc27604595)

[**4.1 Структура синтаксического анализатора** 25](#_Toc27604596)

[**4.2 Контекстно-свободная грамматика, описывающая синтаксис языка** 25](#_Toc27604597)

[**4.3 Построение конечного магазинного автомата** 28](#_Toc27604598)

[**4.4 Основные структуры данных** 29](#_Toc27604599)

[**4.5 Описание алгоритма синтаксического разбора** 29](#_Toc27604600)

[**4.6 Структура и перечень сообщений синтаксического анализатора** 29](#_Toc27604601)

[**4.7. Параметры синтаксического анализатора и режимы его работы** 30](#_Toc27604602)

[**4.8. Принцип обработки ошибок** 30](#_Toc27604603)

[**4.9. Контрольный пример** 30](#_Toc27604604)

[**5 Разработка семантического анализатора** 30](#_Toc27604605)

[**5.1 Структура семантического анализатора** 30](#_Toc27604606)

[**5.2 Функции семантического анализатора** 31](#_Toc27604607)

[**5.3 Структура и перечень сообщений семантического анализатора** 31](#_Toc27604608)

[**5.4 Принцип обработки ошибок** 31](#_Toc27604609)

[**5.5 Контрольный пример** 31](#_Toc27604610)

[**6 Вычисление выражений** 32](#_Toc27604611)

[**6.1 Выражения, допускаемые языком** 32](#_Toc27604612)

[**6.2 Польская запись и принцип её построения** 32](#_Toc27604613)

[**6.3 Программная реализация обработки выражений** 33](#_Toc27604614)

[**6.4 Контрольный пример** 33](#_Toc27604615)

[**7. Генерация кода** 33](#_Toc27604616)

[**7.1 Структура генератора кода** 33](#_Toc27604617)

[**7.2 Представление типов данных в оперативной памяти** 34](#_Toc27604618)

[**7.3 Статическая библиотека** 34](#_Toc27604619)

[**7.4 Особенности алгоритма генерации кода** 35](#_Toc27604620)

[**7.5 Входные параметры генератора кода** 35](#_Toc27604621)

[**7.6 Контрольный пример** 35](#_Toc27604622)

[**8. Тестирование транслятора** 36](#_Toc27604623)

[**8.1 Общие положения** 36](#_Toc27604624)

[**8.2 Результаты тестирования** 36](#_Toc27604625)

[**Заключение** 37](#_Toc27604626)

[**Список использованных источников** 39](#_Toc27604627)

[**Приложение А** 40](#_Toc27604628)

[**Приложение Б** 42](#_Toc27604630)

[**Приложение В** 44](#_Toc27604631)

[**Приложение Г** 48](#_Toc27604632)

[**Приложение Д** 51](#_Toc27604633)

**Введение**

Целью курсового проекта является разработка компилятора для своего языка программирования: ODU-2022. Этот язык программирования предназначен для выполнения простейших операций и арифметических действий над числами.

Компилятор – это программа, задачей которого является перевод программы, написанной на одном из языков программирования в программу на язык ассемблера.

Компиляция состоит из двух частей: анализа и синтеза. Анализ – это разбиение исходной программы на составные части и создание ее промежуточного представления. Синтез – конструирование требуемой целевой программы из промежуточного представления. В данном курсовом проекте мой исходный код транслируется на язык ассемблера. Мой компилятор состоит из следующих составных частей:

– лексический и семантический анализаторы;

– синтаксический анализатор;

– генератор исходного кода на языке 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;

– генерация кода на язык ассемблера;

– 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 и семантика языка.

Во второй главе работы представлена структура транслятора, т.е. перечислены компоненты транслятора, их назначение и принципы взаимодействия, перечень входных параметров, перечень протоколов, формируемых транслятором и их содержимое.

В третьей главе работы показана разработка лексического анализатора, порождающего таблицы лексем и 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, показана его работа (распечатка выданных сообщений в трёх примерах на разных этапах).

В шестой главе решены вопросы преобразования выражений, допускаемых языком и приведена часть протокола для контрольного примера, отображающая результаты преобразования выражений в польский формат.

В седьмой главе представлена генерация кода, где из промежуточного представления порождается код на целевом языке.

В восьмой главе описывается тестирование транслятора

# **1. Спецификация языка программирования**

## **Характеристика языка программирования**

Язык программирования ODU-2022 является процедурным, строго типизированным, не объектно-ориентированным, компилируемым.

Процедурный язык программирования — язык высокого уровня, в котором используется метод разбиения программ на отдельные связанные между собой модули — подпрограммы (процедуры и функции).

* 1. **Определение алфавита языка программирования**

В алфавите языка ODU-2022 используется кодировка ASCII, таблица которой

представлена на рисунке 1.1.



Рисунок 1.1 – Алфавит входных символов

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

* 1. **Применяемые сепараторы**

Применяемые сепараторы в языке ODU-2022, приведены в таблице 1.1.

Таблица 1.1 — Применяемые сепараторы

|  |  |
| --- | --- |
| Сепаратор | Область применения |
| пробел | Разделитель цепочек. Допускается везде кроме названий идентификаторов и ключевых слов |

Продолжение таблицы 1.1

|  |  |
| --- | --- |
| Сепаратор | Область применения |
| ; | Разделитель программных конструкций |
| { } | Блок функции |
| = | Оператор присваивания |
| ( ) | Блок фактических или формальных параметров функции, а также приоритет арифметических операций |
| , | Разделитель параметров функций |
| +,-,\*,/ ,% | Арифметические операции |
| [] | Блок условной конструкции/цикла |
| == <= >= != > < | Логические операции (операции сравнения: равно, не равно, больше, меньше, больше либо равно, меньше либо равно), используемые в условии цикла/условной конструкции. |

## **Применяемые кодировки**

Для написания программ язык ODU-2022 использует кодировку ASCII, содержащую английский алфавит, а также некоторые специальные символы, такие как { } [ ] ( ) , ; ~ + - / \* > < & !.

ASCII (American standard code for information interchange) — название таблицы (кодировки, набора), в которой некоторым распространённым печатным и непечатным символам сопоставлены числовые коды. Таблица была разработана и стандартизована в [США](https://ru.wikipedia.org/wiki/%D0%A1%D0%BE%D0%B5%D0%B4%D0%B8%D0%BD%D1%91%D0%BD%D0%BD%D1%8B%D0%B5_%D0%A8%D1%82%D0%B0%D1%82%D1%8B_%D0%90%D0%BC%D0%B5%D1%80%D0%B8%D0%BA%D0%B8), в 1963 году.

## **Типы данных**

В языке ODU-2022 есть 3 типа данных: знаковый целочисленный, строковый и символьный. Описание типов данных, предусмотренных в данным языке представлено в таблице 1.2.

Таблица 1.2 – Типы данных языка ODU-2022

|  |  |
| --- | --- |
| Тип данных | Описание типа данных |
| int (integer) | Фундаментальный тип данных. Используется для работы с числовыми значениями. В памяти занимает 4 байта. Максимальное значение: 2147483647.  Минимальное значение: -2147483647.  Инициализация по умолчанию: значение 0.  Поддерживаемые операции:  + (бинарный) – оператор сложения;  - (бинарный) – оператор вычитания;  \* (бинарный) – оператор умножения; |

Продолжение таблицы 1.2

|  |  |
| --- | --- |
| Тип данных | Описание типа данных |
|  | / (бинарный) – оператор деления;  %(бинарный) – оператор остаток от деления;  = (бинарный) – оператор присваивания.  В качестве условия цикла/условного оператора поддерживаются следующие логические операции:  > (бинарный) – оператор «больше»  < (бинарный) – оператор «меньше»;  >= (бинарный) – оператор «больше либо равно»;  <= (бинарный) – оператор «меньше либо равно»;  == (бинарный) – оператор проверки на равенство;  != (бинарный) – оператор проверки на неравенство. |
| string | Фундаментальный тип данных. Используется для работы с символами, каждый из которых занимает 1 байт. Максимальное количество символов – 255.  Инициализация по умолчанию: строка нулевой длины “”. Операции над данными строкового типа: присваивание строковому идентификатору значения другого строкового идентификатора, строкового литерала или значения строковой функции, а также использование библиотечных функций. |
| char | Фундаментальный тип данных. Используется для работы с символами, каждый из которых занимает 1 байт. Максимальное количество символов – 1.  Инициализация по умолчанию: символ нулевой длины “”. Операции над данными строкового типа: присваивание символьному идентификатору значения другого символьного идентификатора, символьного литерала или значения символьной функции, а также использование библиотечных функций. |
| bool | Фундаментальный тип данных. Используется для работы с числовыми значениями. В памяти занимает 1 байт.  Может принимать лишь TRUE (1) либо FALSE (0).  Инициализация по умолчанию: значение FALSE.  Поддерживаемые операции:  == (бинарный) – оператор проверки на равенство;  != (бинарный) – оператор проверки на неравенство. |

## **Преобразование типов данных**

В языке программирования ODU-2022 преобразование типов данных не поддерживается, т.е. язык является строго типизированным.

## **Идентификаторы**

Общее количество идентификаторов ограничено максимальным размером таблицы идентификаторов. Идентификаторы должны начинаться только с символов латинского алфавита, могут содержать цифры. Максимальная длина идентификатора равна восьми символам. Идентификаторы, объявленные внутри функционального блока, получают префикс, идентичный имени функции, внутри которой они объявлены. Префикс занимает 8 дополнительных символов. В случае превышения заданной длины, идентификаторы усекаются до длины, равной 16 символов (8 символов на имя идентификатора, 8 символов на префикс). Данные правила действуют для всех типов идентификаторов. Идентификаторы не должны совпадать с ключевыми словами.

## **Литералы**

С помощью литералов осуществляется инициализация переменных. Все литералы являются rvalue. Типы литералов языка ODU-2022 представлены в таблице 1.3.

Таблица 1.3 – Описание литералов

|  |  |
| --- | --- |
| Тип литерала | Описание |
| Литералы целого типа(десятичного представления) | Последовательность цифр 0…9 с предшествующим знаком минус или без него (знак минус не должен отделяться пробелом) |
| Литералы целого типа(восьмеричного представления) | Последовательность цифр 0…7 начиная с 0 с предшествующим знаком минус или без него (знак минус не должен отделяться пробелом) |
| Строковые литералы | Набор символов (от 1 до 255), заключённых в “ ” |
| Символьные литералы | Символ , заключенный в ~ ~ |
| Логические литералы | Два состояния: TRUE, FALSE |

## **Объявление данных**

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

Пример объявления числового типа с инициализацией:

**type integer** num1 = -1

Пример объявления переменной строкового типа с инициализацией:

**type string** str1 = “hello world”

Пример объявления переменной символьного типа с инициализацией:

**type symbol** ch1 = ~c~

Пример объявления логического типа с инициализацией:

**type bool** bl1 = true;

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

## **Инициализация данных**

При объявлении переменной допускается инициализация данных. При этом переменной будет присвоено значение литерала или идентификатора, стоящего справа от знака равенства. При объявлении без инициализации предусмотрены значения по умолчанию: значение 0 для типа **integer** и строка длины 0 (“”) для типа **string** и **symbol**.

## **Инструкции языка**

Инструкции языка ODU-2022 представлена в таблице 1.4.

Таблица 1.4 – Инструкции языка

|  |  |
| --- | --- |
| Инструкция | Форма записи |
| Объявление переменной | **type** <тип данных> <идентификатор>. |
| Объявление переменной с явной инициализацией | **type** <тип данных> <идентификатор> = <значение>|<выражение>;  Значение – литерал, идентификатор, вызов функции соответствующего типа данных |
| Объявление функции | <тип данных> **function** <идентификатор> (<тип данных> <идентификатор>, …)  {  / программный блок /  **return** <идентификатор/литерал>.  } |
| Вызов функции | <идентификатор> (<идентификатор>, ...) |
| Присвоение значения | <идентификатор> = <значение>; |
| Вывод данных | **write** <идентификатор/литерал>; |
| Вывод данных с переходом на новую строку | **writeline** <идентификатор/литерал>; |
| Возвращаемое значение | **return** <литерал/идентификатор>. |

## **Операции языка**

В языке ODU-2022 предусмотрены следующие операции с данными. Приоритетность операции умножения выше приоритета операций сложения и вычитания. Для установки наивысшего приоритета используются круглые скобки. Операции языка представлены в таблице 1.5.

Таблица 1.5 — Операции языка

|  |  |
| --- | --- |
| Тип оператора | Оператор |
| Арифметические операции языка | + – сложение  - – вычитание |

Продолжение таблицы 1.5

|  |  |
| --- | --- |
| Тип оператора | Оператор |
|  | \* – умножение  / – деление  % – остаток от деления  = – присваивание |
| Логические | > – больше  < – меньше  >= – больше либо равно  <= – меньше либо равно  == – проверка на равенство  != – проверка на неравенство |

## **Выражение и их вычисление**

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

* допускается использовать скобки для смены приоритета операций;
* выражение записывается в строку без переносов;
* использование двух подряд идущих операторов не допускается. (за исключением оператор “-”, идущего после любого другого оператора и предшествующему литералу);
* допускается использовать в выражении вызов функции, вычисляющей и возвращающей целочисленное значение.

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

## **Конструкции языка**

Программа на языке ODU-2022 оформляется в виде функций пользователя и главной функции. При составлении функций рекомендуется выделять блоки и фрагменты и применять отступы для лучшей читаемости кода.

Программные конструкции языка представлены в таблице 1.6

Таблица 1.6 — Основные конструкции языка

|  |  |
| --- | --- |
| Конструкция | Реализация |
| Главная функция | main  {  …} |
| Внешняя функция | <тип данных> function <идентификатор> (<тип> <идентификатор>, ...)  {…  return <идентификатор/литерал>;  } |

Продолжение таблицы 1.6

|  |  |  |
| --- | --- | --- |
| Конструкция | | Реализация |
| Условная конструкция | **if** <идентификатор1> <оператор> <идентификатор2>  **then** [ … ]  **else** [ … ]  <идентификатор1>, <идентификатор2> - идентификаторы или литералы. <оператор> - один из операторов сравнения ( == != > < >= <=), устанавливающий отношение между двумя операндами и организующий условие данной конструкции. Причем с идентификаторами и литералами строкового и символьного типа, могут использоваться только операторы сравнения (& !) При истинности условия выполняется код внутри блока **then**, иначе – код внутри блока **else**. Блок **else** может отсутствовать. | |
| Цикл | **while** <идентификатор1> <оператор> <идентификатор2>  **do** [ … ]  Цикл (операторы внутри блока do) выполняется, пока истинно условие “<идентификатор1> <оператор> <идентификатор2>”, имеющее тот же смысл, что и в примере выше. | |

## **Область видимости переменных**

Область видимости: сверху вниз (как и в С++). Переменные, объявленные в одной функции, недоступны в другой. Все объявления и операции с переменными происходят внутри какого-либо блока. Каждая переменная или параметр функции получают префикс – название функции, внутри которой они находятся.

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

## **Семантические проверки**

* В языке программирования ODU-2022 выполняются следующие семантические проверки:
* наличие функции main – точки входа в программу;
* единственность точки входа;
* переопределение идентификаторов;
* использование идентификаторов без их объявления;
* проверка соответствия типа функции и возвращаемого параметра;
* правильность передаваемых в функцию параметров: количество, типы;
* правильность строковых выражений;
* превышение размера строковых и числовых литералов;
* правильность составленного условия цикла/условного оператора.

## **Распределение оперативной памяти на этапе выполнения**

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

## **Стандартная библиотека и её состав**

В языке ODU-2022 предусмотрена стандартная библиотека. Функции, входящие в состав библиотеки, описаны в табл. 1.7. Стандартная библиотека подключается автоматически на этапе генерации кода.

Таблица 1.7 - Функции стандартной библиотеки языка ODU-2022

|  |  |
| --- | --- |
| Функция | Описание |
| string concat(string str1, string str2) | Функция конкатенирует две строки в одну и возвращает конкатенированную строку. |
| string copy(string str) | Функция создаёт копию передаваемой строки и возвращает эту копию. |
| int random(int a) | Данная функция целочисленного типа принимает 1 параметр. Функция генерирует случайное число в диапазоне от -a до а и возвращает это число. |
| int pow (int a, int b) | Данная функция целочисленного типа принимает два целочисленных параметра. Функция возводит число a в степень b и возвращает результат. |
| int length (string a) | Данная функция целочисленного типа принимает 1 параметр.  Функция возвращает длину строки a. |
| String intToStr(int a) | Функция преобразует переданный целочисленный тип в строку и возвращает её. |
| int inputInt() | Функция для вывода из стандартного потока. Введённые данные из консоли будут интерпретированы как целочисленные. Возвращает введённые данные. |
| Char inputChar() | Функция для вывода из стандартного потока. Возвращает введённые данные в виде строки. |
| string inputString() | Функция для вывода из стандартного потока. Запрашивает ввод одного символа, возвращает этот символ. |

Стандартная библиотека написана на языке С++, подключается к транслированному коду на этапе генерации кода. Вызовы стандартных функций доступны там же, где и вызов пользовательских функций. Также в стандартной библиотеке реализованы функции для манипулирования выводом, недоступные конечному пользователю. Для вывода предусмотрен оператор **Write** и **WriteLine**. Эти функции представлены в таблице 1.8.

Таблица 1.8 Дополнительные функции стандартной библиотеки

|  |  |
| --- | --- |
| Функция на языке С++ | Описание |
| void outnum(int value) | Функция для вывода в стандартный поток значения целочисленного идентификатора/литерала. |
| void outstr(char\* line) | Функция для вывода в стандартный поток значения строкового, символьного идентификатора/литерала. |
| void outstrline(char\* line) | Функция для вывода в стандартный поток значения строкового, символьного идентификатора/литерала и перехода на новую строку. |
| void outnumline(int value) | Функция для вывода в стандартный поток значения целочисленного идентификатора/литерала и перехода на новую строку. |
| void system\_pause() | Функция которая запускается по окончанию программы и требует нажать любой символ для завершения программы |

## **Ввод и вывод данных**

Вывод данных осуществляется с помощью операторов **Write** и **WriteLine**. Допускается использование операторов **Write** и **WriteLine** с литералами и идентификаторами.

Функции, управляющие выводом данных, реализованы на языке С++ и вызываются из транслированного кода, конечному пользователю недоступны. Пользовательские команды **Write** и **WriteLine** в транслированном коде будут заменены вызовом нужных библиотечных функций. Библиотека, содержащая нужные процедуры, подключается на этапе генерации кода.

## **Точка** **входа**

В языке ODU-2022 каждая программа должна содержать главную функцию (точку входа) **Main**, с первой инструкции которой начнётся последовательное выполнение команд программы.

## **Препроцессор**

Препроцессор, принимающий и выдающий некоторые данные на вход транслятору, в языке ODU-2022 отсутствует.

## **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.

## **Объектный код**

ODU-2022 транслируется в язык ассемблера, а затем - в объектный код

## **Классификация сообщений транслятора**

Генерируемые транслятором сообщения определяют степень его информативности, то есть сообщения транслятора должны давать максимально полную информацию о допущенной пользователем ошибке при написании программы. Сообщения транслятора приведены в таблице 1.9, а также в приложении A.

Таблица 1.9 Классификация ошибок

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

## **Контрольный пример**

Контрольный пример демонстрирует главные особенности языка ODU-2022: его фундаментальные типы, основные структуры, функции, процедуры, использование функций статической библиотеки. Исходный код контрольного примера представлен в приложении А.

# **2. Структура транслятора**

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

Транслятор преобразует программу, написанную на языке ODU-2022 в программу на языке ассемблера. Компонентами транслятора являются лексический, синтаксический и семантический анализаторы, а также генератор кода на язык ассемблера. Принцип их взаимодействия представлен на рисунке 2.1.



Рисунок 2.1 Структура транслятора языка программирования ODU-2022

Лексический анализ – первая фаза трансляции. Назначением лексического анализатора является нахождение ошибок лексики языка и формирование таблицы лексем и таблицы идентификаторов. Подробнее описан в 3 главе.

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

Семантический анализ в свою очередь является проверкой исходной программы на семантическую согласованность с определением языка, т.е. проверяет правильность текста исходной программы с точки зрения семантики. Подробное описание представлено в 5 главе.

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 ODU-2022, прошедший все предыдущие этапы, в код на языке Ассемблера. Более полно описан в главе 7.

## **2.2 Перечень входных параметров транслятора**

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

Таблица 2.1 Входные параметры транслятора языка ODU-2022

|  |  |  |
| --- | --- | --- |
| Входной параметр | Описание параметра | Значение по умолчанию |
| -in:<путь к in-файлу> | Файл с исходным кодом на языке ODU-2022, имеющий расширение .txt | Не предусмотрено |

Продолжение таблицы 2.1

|  |  |  |
| --- | --- | --- |
| Входной параметр | Описание параметра | Значение по умолчанию |
| -log:<путь к log-файлу> | Файл журнала для вывода протоколов работы программы. | Значение по умолчанию:  <имя in-файла>.log |
| -out:<путь к out-файлу> | Выходной файл – результат работы транслятора. Содержит исходный код на языке асемблера. | Значение по умолчанию:  <имя in-файла>.asm |

## **2.3 Протоколы, формируемые транслятором**

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

Таблица 2.2 Протоколы, формируемые транслятором языка ODU-2022

|  |  |
| --- | --- |
| Формируемый протокол | Описание выходного протокола |
| Файл журнала, заданный параметром "-log:" | Файл с протоколом работы транслятора языка программирования ODU-2022. 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, заданный параметром "-out:" | Результат работы программы – файл, содержащий исходный код на языке ассемблера. |

# **3** **Разработка лексического анализатора**

## **3.1 Структура лексического анализатора**

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

Примеры лексических единиц: идентификаторы, числа, символы операций, служебные слова и т.д. Лексический анализатор преобразует исходный текст, заменяя лексические единицы их внутренним представлением – лексемами, для создания промежуточного представления исходной программы. Каждой лексеме сопоставляется ее тип и запись в таблице идентификаторов, в которой хранится дополнительная информация.

Функции лексического анализатора:

− удаление «пустых» символов и комментариев. Если «пустые» символы (пробелы, знаки табуляции и перехода на новую строку) и комментарии будут удалены лексическим анализатором, синтаксический анализатор никогда не столкнется с ними (альтернативный способ, состоящий в модификации грамматики для включения «пустых» символов и комментариев в синтаксис, достаточно сложен для реализации);

− распознавание идентификаторов и ключевых слов;

− распознавание констант;

− распознавание разделителей и знаков операций.

Исходный код программы представлен в приложении А, структура лексического анализатора представлена на рисунке 3.1.



Рисунок 3.1 Структура лексического анализатора

## **3.2 Контроль входных символов**

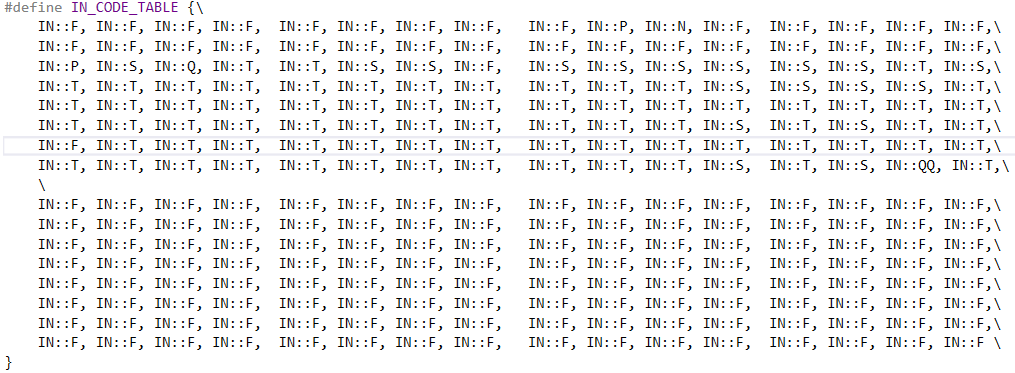
Для удобной работы с исходным кодом, при передаче его в лексический анализатор, все символы разделяются по категориям. Таблица входных символов представлена на рисунке 3.2, категории входных символов представлены в таблице 3.1.

Рисунок 3.2. Таблица контроля входных символов

Таблица 3.1 Соответствие символов и их значений в таблице

|  |  |
| --- | --- |
| Значение в таблице входных символов | Символы |
| Разрешенный | T |
| Запрещенный | F |
| Игнорируемый | I |
| Строковый литерал | Q |
| Символьный литерал | QQ |
| Сепаратор | S |
| Перевод строки | N |
| Пробел, табуляция | P |

## **3.3 Удаление избыточных символов**

Избыточными символами являются символы табуляции и пробелы.

Избыточные символы удаляются на этапе разбиения исходного кода на токены.

Описание алгоритма удаления избыточных символов:

1. Посимвольно считываем файл с исходным кодом программы;

2. Встреча пробела или знака табуляции является своего рода встречей символа-сепаратора;

3. В отличие от других символов-сепараторов не записываем в очередь лексем эти символы, т.е. игнорируем.

## **3.4 Перечень ключевых слов**

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

Таблица 3.2 Соответствие токенов и сепараторов с лексемами

|  |  |  |
| --- | --- | --- |
| Токен | Лексема | Пояснение |
| integer, string,symbol | t | Названия типов данных языка. |
| Идентификатор | i | Длина идентификатора – 8 символов. |
| Литерал | l | Литерал любого доступного типа. |
| function | f | Объявление функции. |
| Функции стандартной библиотеки | p | Ключевое слово для стандартных функций. |
| return | e | Выход из функции/процедуры. |
| main | m | Главная функция. |
| type | n | Объявление переменной. |
| write | o | Вывод данных. |
| writeline | ^ | Вывод данных c переводом строки. |
| if | ? | Указывает начало условного оператора. |
| while | c | Указывает начало цикла. |
| then | r | Истинная ветвь условного оператора. |
| else | w | Ложная ветвь условного оператора. |
| do | d | Указывает на начало тела цикла. |
| ; | ; | Разделение выражений. |
| , | , | Разделение параметров функций. |
| [ | [ | Начало блока цикла/условного выражения. |
| ] | ] | Конец блока цикла/условного выражения. |
| { | { | Начало тела функции. |
| } | } | Конец тела функции |
| ( | ( | Передача параметров в функцию, приоритет операций. |
| ) | ) | Закрытие блока для передачи параметров, приоритет операций. |
| = | = | Знак присваивания. |
| +  -  \*  /  % | +  -  \*  /  % | Знаки операций. |

Продолжение таблицы 3.2

|  |  |  |
| --- | --- | --- |
| Токен | Лексема | Пояснение |
| ==  !=  >  <  >=  <= | 0  1  2  3  4  5 | Знаки логических операторов |

Пример реализации таблицы лексем представлен в приложении Б.

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

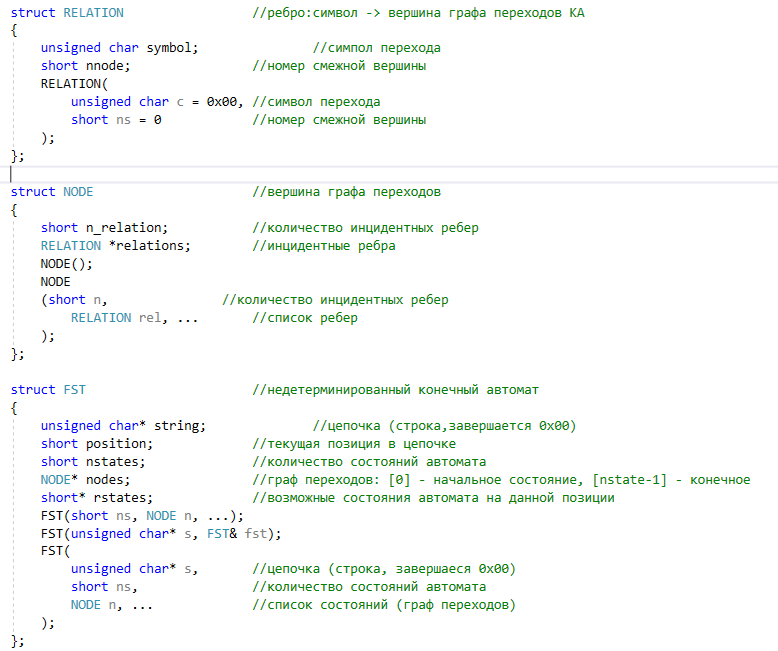


Рисунок 3.3 Структура конечного автомата

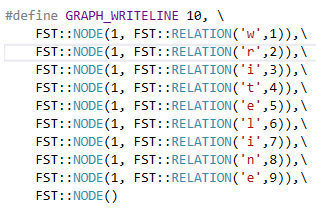


Рисунок 3.4 Реализация графа конечного автомата для токена writeline

## **3.5 Основные структуры данных**

Основными структурами данных лексического анализатора являются таблица лексем и таблица идентификаторов. Таблица лексем содержит номер лексемы, лексему (lexema), полученную при разборе, номер строки в исходном коде (sn), и номер в таблице идентификаторов, если лексема является идентификаторов (idxTI). Таблица идентификаторов содержит имя идентификатора (id), номер в таблице лексем (idxfirstLE), тип данных (iddatatype), тип идентификатора (idtype) и значение (или параметры функций) (value). Код C++ со структурой таблицы лексем представлен на рисунке 3.3. Код C++ со структурой таблицы идентификаторов представлен на рисунке 3.4.

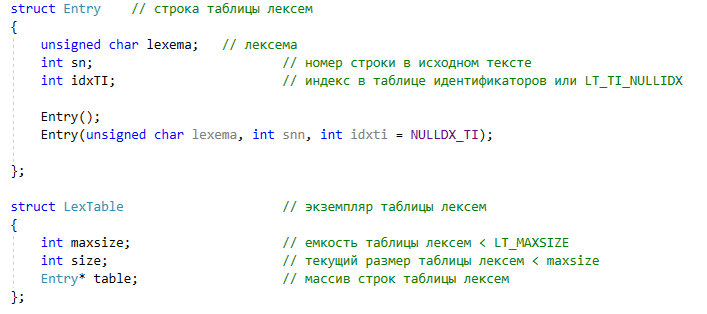


Рисунок 3.3 Структура таблицы лексем

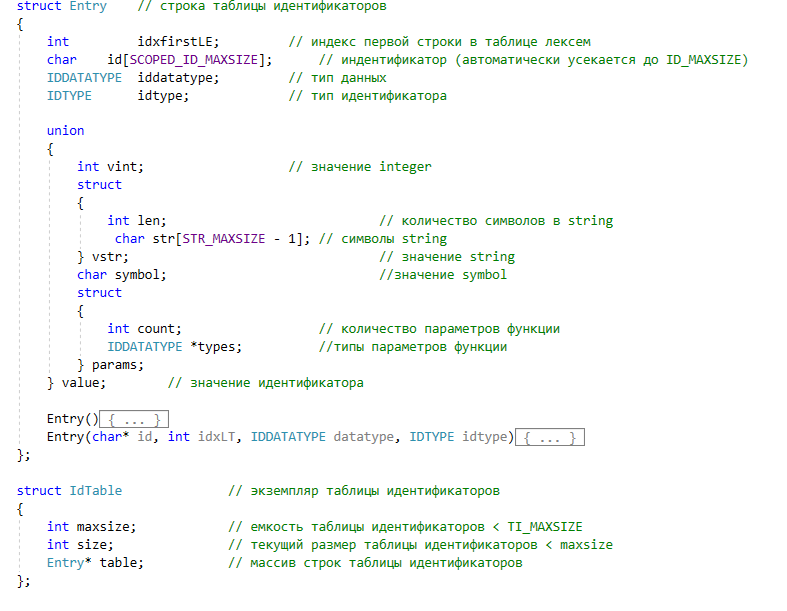


Рисунок 3.4 Структура таблицы идентификаторов

## **3.6 Принцип обработки ошибок**

Для обработки ошибок лексический анализатор использует таблицу с сообщениями. Структура сообщений содержит информацию о номере сообщения, номер строки и позицию, где было вызвано сообщение в исходном коде, информацию об ошибке. При возникновении сообщения, лексический анализатор игнорирует найденную ошибку и продолжает работу с исходным кодом. Перечень сообщений представлен на рисунке 3.5.



Рисунок 3.5 - Сообщения лексического анализатора

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

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. Если в процессе анализа находятся более трёх ошибок, то анализ останавливается.

## **3.8 Параметры лексического анализатора**

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**

* проверяет входной поток символов программы на исходном языке на допустимость, удаляет лишние пробелы и добавляет сепаратор для вычисления номера строки для каждой лексемы;
* для выделенной части входного потока выполняется функция распознавания лексемы;
* при успешном распознавании информация о выделенной лексеме заносится в таблицу лексем и таблицу идентификаторов, и алгоритм возвращается к первому этапу;
* формирует протокол работы;
* при неуспешном распознавании выдается сообщение об ошибке.
* Распознавание цепочек основывается на работе конечных автоматов. Работу конечного автомата можно проиллюстрировать с помощью графа переходов. Пример графа для цепочки «string» представлен на рисунке 3.2, где S0 – начальное, а S6 – конечное состояние автомата.

Распознавание цепочек основывается на работе конечных автоматов. Работу конечного автомата можно проиллюстрировать с помощью графа переходов. Пример графа для цепочки «string» представлен на рисунке 3.2, где S0 – начальное, а S6 – конечное состояние автомата.

s t r i n g

Рисунок 3.2 Пример графа переходов для цепочки string

## **3.10 Контрольный пример**

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

# **4. Разработка синтаксического анализатора**

## **4.1 Структура синтаксического анализатора**

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

Описание структуры синтаксического анализатора языка представлено на рисунке 4.1.



Рисунок 4.1 Структура синтаксического анализатора.

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

В синтаксическом анализаторе транслятора языка ODU-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, т.к. она не леворекурсивная (не содержит леворекурсивных правил) и правила  имеют вид:

1. , где ; (или , или );
2. , где — начальный символ, при этом если такое правило существует, то нетерминал  не встречается в правой части правил.

Описание нетерминальных символов содержится в таблице 4.1.

Таблица 4.1 Таблица правил переходов нетерминальных символов

|  |  |  |
| --- | --- | --- |
| Символ | Правила | Какие правила порождает |
| S | S->tfiPTS  S->m{K} | Стартовые правила, описывающее общую структуру программы |
| P | P->(E)  P->() | Правила для параметров объявляемых функций |
| T | T->{Q;}  T->[KQ;] | Правила для тела функций |
| Q | Q->eV;  Q->e(l);  Q->e;  Q->e(-i); | Правила для возврата значений |
| E | E->ti,E  E->ti | Правила для списка параметров функции |
| F | F->(N)  F->() | Правила для вывозов функций(в т.ч. и в выражениях) |
| N | N->i  N>l  N->l,N  N->i,N | Правила для параметров вызываемых функций |
| R | R->rY  R->wY  R->rYwY  R->wYrY | Правила составления условного оператора |
| Z | Z->iLi  Z->iLl  Z->lli | Правила для условия цикла/условного оператора |
| L | L->0  L->1  L->2  L->4  L->5 | Правила для логических операторов |
| A | A->+  A->-  A->\*  A->/  A->% | Правила для арифметических операторов |

Продолжение таблицы 4.1

|  |  |  |
| --- | --- | --- |
| Символ | Правила | Какие правила порождает |
| V | V->l  V->i | Правила для простых выражений |
| Y | Y->[XQ]  Y->[X]  Y->[Q]  Y->[] | Правила для тела условного выражения |
| W | W->l  W->i  W->(W)  W->(-i)  W->(-i)AW  W->(W)AW  W->iF  W->pF  W->iAW  W->lAW  W->iFAW  W->pFAW | Правила для сложных выражений |
| K | K->nti=V;K  K->nti=WK  K->nti;K  K->i=W;K  K->oV;K  K->^VZ;K  K->?ZdHK  K->cZdHK  K->i=UF;K  K->nti=V;  K->nti=W;  K->i=W;  K->nti;  K->oV;  K->^V;  K->?ZR  K->cZdH  K->i=UF; | Программные конструкции |

Продолжение таблицы 4.1

|  |  |  |
| --- | --- | --- |
| X | X->i=W;X  X->oV;X  X->^V;X  X->i=UF;X  X->i=W;  X->oV;  X->^V;  X->i=UF; | Программные конструкции внутри цикла/условного оператора |
| U | U->i  U->p | Правила для идентификатора функции |
| B | B->?ZR  B->?ZRX  B->?ZRXB  B->?ZRB | Правила для конструирования условного выражения в цикле |
| H | H->[XQ]  H->[Q]  H->[X]  H->[] | Правила для тела цикла |

## 

## **4.3 Построение конечного магазинного автомата**

Конечный автомат с магазинной памятью представляет собой семерку. Подробное описание компонентов магазинного автомата представлено в таблице 4.2.

Таблица 4.2 – Описание компонентов магазинного автомата

|  |  |  |
| --- | --- | --- |
| Компонента | Определение | Описание |
|  | Множество состояний автомата | Состояние автомата представляет из себя структуру, содержащую позицию на входной ленте, номера текущего правила и цепочки и стек автомата |
|  | Алфавит входных символов | Алфавит представляет из себя множества терминальных и нетерминальных символов, описание которых содержится в таблица 3.1 и 4.1. |
|  | Алфавит специальных магазинных символов | Алфавит магазинных символов содержит стартовый символ и маркер дна стека (представляет из себя символ $) |
|  | Функция переходов автомата | Функция представляет из себя множество правил грамматики, описанных в таблице 4.1. |

Продолжение таблицы 4.2

|  |  |  |
| --- | --- | --- |
| Компонента | Определение | Описание |
|  | Начальное состояние автомата | Состояние, которое приобретает автомат в начале своей работы. Представляется в виде стартового правила грамматики |
|  | Начальное состояние магазина автомата | Символ маркера дна стека $ |
|  | Множество конечных состояний | Конечные состояние заставляют автомат прекратить свою работу. Конечным состоянием является пустой магазин автомата и совпадение позиции на входной ленте автомата с размером ленты |

## **4.4 Основные структуры данных**

Основные структуры данных синтаксического анализатора представляются в виде структуры магазинного конечного автомата, выполняющего разбор исходной ленты, и структуры грамматики Грейбах, описывающей синтаксические правила языка ODU-2022. Правила языка ODU-2022 в приложении В.

## **4.5 Описание алгоритма синтаксического разбора**

Принцип работы автомата следующий:

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

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

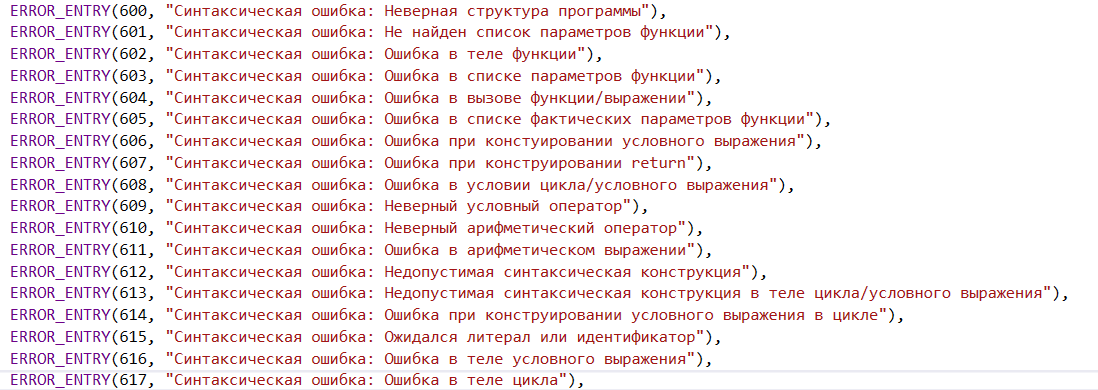
 Список сообщений синтаксического анализатора представлен на рисунке 4.3.

Рисунок 4.3 - Сообщения синтаксического анализатора

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

Входной информацией для синтаксического анализатора является таблица лексем и идентификаторов. Кроме того, используется описание грамматики в форме Грейбах. Результаты работы синтаксического разбора, а именно дерево разбора и протокол работы автомата с магазинной памятью выводятся в журнал работы программы.

## **4.8. Принцип обработки ошибок**

Синтаксический анализатор выполняет разбор исходной последовательности лексем до тех пор, пока не дойдёт до конца цепочки лексем или не найдёт ошибку. Тогда анализ останавливается и выводится сообщение об ошибке (если она найдена). Если в процессе анализа находятся более трёх ошибок, то анализ останавливается.

## **4.9. Контрольный пример**

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

# **5 Разработка семантического анализатора**

## **5.1 Структура семантического анализатора**

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



Рисунок 5.1. Структура семантического анализатора

## **5.2 Функции семантического анализатора**

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

## **5.3 Структура и перечень сообщений семантического анализатора**

Сообщения, формируемые семантическим анализатором, представлены на рисунке 5.2.

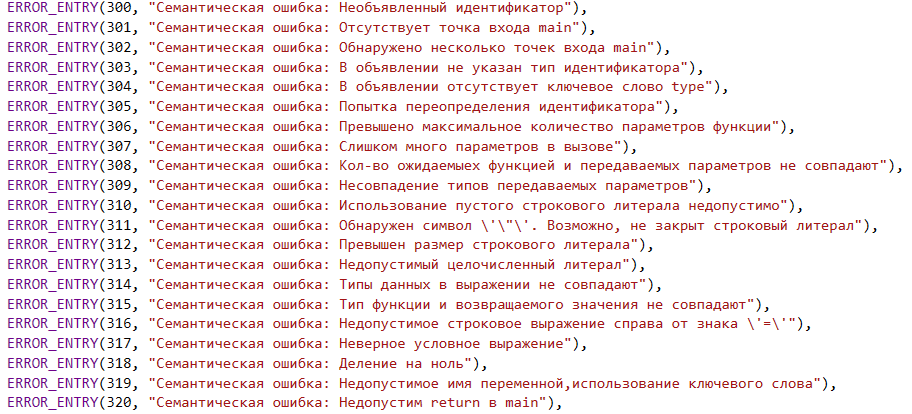


Рисунок 5.2 – Перечень сообщений семантического анализатора

## **5.4 Принцип обработки ошибок**

Ошибки, возникающие в процессе трансляции программы, фиксируются в протокол, заданный входным параметрами. В случае возникновения ошибок происходит их протоколирование с номером ошибки и диагностическим сообщением. Анализ останавливается после того, как будут найдены все ошибки.

## **5.5 Контрольный пример**

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

Таблица 5.1. Примеры диагностики ошибок

|  |  |
| --- | --- |
| Исходный код | Текст сообщения |
| main{  integer x = 9;  write x;  } | Ошибка N304: Семантическая ошибка: В объявлении отсутствует ключевое слово type Строка: 2 |
| main{  type integer x = 9;  type integer y =x;  } | Ошибка N314: Семантическая ошибка: Типы данных в выражении не совпадают Строка: 3 |

Продолжение таблицы 5.1

|  |  |
| --- | --- |
| Исходный код | Текст сообщения |
| main{  type integer x = 9;  }  main{  type string y = "qwerty";  } | Ошибка N302: Семантическая ошибка: Обнаружено несколько точек входа main |

# **6 Вычисление выражений**

## **6.1 Выражения, допускаемые языком**

В языке ODU-2022 допускаются вычисления выражений целочисленного типа данных с поддержкой вызова функций внутри выражений. Приоритет операций представлен на таблице 6.1.

Таблица 6.1. Приоритеты операций

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

## **6.2 Польская запись и принцип её построения**

Все выражения языка ODU-2022 преобразовываются к обратной польской записи.

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

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

## **6.3 Программная реализация обработки выражений**

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

## **6.4 Контрольный пример**

Пример преобразования выражения из контрольного примера к обратной польской записи представлен в таблице 6.2. Преобразование выражения в формат польской записи необходимо для построения более простых алгоритмов их вычисления и преобразования к ассемблерному коду. В приложении Г приведены изменённые таблицы лексем и идентификаторов, отображающие результаты преобразования выражений в польский формат.

Таблица 6.2. Преобразование выражения к ПОЛИЗ

|  |  |
| --- | --- |
| Выражение | Обратная польская запись для выражения |
| i[27]=(-i[20])+(i[24]+i[0](1[7]))\*1[28] | i[27]=i[20]~i[24]i[0]1[7]@1+1[28]\*+; |

# **7. Генерация кода**

## **7.1 Структура генератора кода**

В языке ODU-2022 генерация кода является заключительным этапом трансляции. Генератор принимает на вход таблицы лексем и идентификаторов, полученные в результате лексического анализа. В соответствии с таблицей лексем строится выходной файл на языке ассемблера, который будет являться результатом работы транслятора. В случае возникновения ошибок генерация кода не будет осуществляться. Структура генератора кода ODU-2022 представлена на рисунке 7.1.



Рисунок 7.1 – Структура генератора кода

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

Элементы таблицы идентификаторов расположены сегментах .data и .const языка ассемблера. Соответствия между типами данных идентификаторов на языке ODU-2022 и на языке ассемблера приведены в таблице 7.1.

Таблица 7.1 – Соответствия типов идентификаторов языка и языка ассемблера

|  |  |  |
| --- | --- | --- |
| Тип идентификатора на языке ODU-2022 | Тип идентификатора на языке ассемблера | Пояснение |
| int | sdword | Хранит целочисленный тип данных. |
| string, char, bool | dword | Хранит указатель на начало строки. Строка должна завешаться нулевым символом.  Хранит ссылку на bool. |

## **7.3 Статическая библиотека**

В языке ODU-2022 предусмотрена статическая библиотека. Статическая библиотека содержит функции, написанные на языке C++. Объявление функций статической библиотеки генерируется автоматически в коде ассемблера. Объявление функций статической библиотеки генерируется автоматически.

Таблица 7.3 – Функции статической библиотеки

|  |  |
| --- | --- |
| Функция | Назначение |
| void outstr(char\* ptr) | Вывод на консоль строки ptr |
| void outnum(int value) | Вывод на консоль переменной value |
| void outstrline(char\* ptr) | Вывод на консоль строки ptr и перевод на новую строку |
| void outnumline(int value) | Вывод на консоль целочисленной переменной value и перевод на новую строку |
| int lenght(char\* str) | Вычисление длины строки |
| void system\_pause() | Ожидание нажатия клавиши пользователем |
| int random(int a) | Генерация случайного числа в диапазоне от -a до a |
| int power(int a, int b) | Возведение числа a в степень b |
| char\* concat(char\*, char\*) | Конкатенация двух строк |
| char\* copy(char\* a) | Копирование строки |
| char\* inttostr(int a) | Перевод числа в строку |
| int inputInt() | Ввод числа из консоли |
| char\* inputString() | Ввод строки из консоли |
| char\* inputChar() | Ввод символа из консоли |

## **7.4 Особенности алгоритма генерации кода**

В языке ODU-2022 генерация кода строится на основе таблиц лексем и идентификаторов. Схема работы генератора кода представлена на рисунке 7.2

Рисунок 7.2 – Структура генератора кода

## **7.5 Входные параметры генератора кода**

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

## **7.6 Контрольный пример**

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

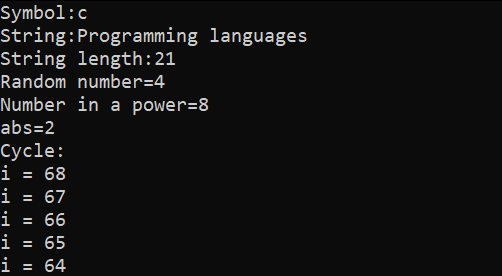


Рисунок 7.2 Результат работы программы на языке ODU-2022

# **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 файл логирования.

## **8.2 Результаты тестирования**

Результаты тестирования приведены в таблице 8.1

Таблица 8.1 – Результаты тестирования транслятора

|  |  |
| --- | --- |
| Исходный код | Диагностическое сообщение |
| мэйн | Ошибка 200: Лексическая ошибка: Недопустимый символ в исходном файле(-in) Строка: 1 |
| main{  type integer i = a;  } | Ошибка 300: Семантическая ошибка: Необъявленный идентификатор Строка: 2 |
| integer function a(){  return 1;  } | Ошибка 301: Семантическая ошибка: Отсутствует точка входа main |
| main{  integer i = 3;  } | Ошибка 304: Семантическая ошибка: В объявлении отсутствует ключевое слово type Строка: 2 |
| main{type integer i;  type integer i;} | Ошибка 305: Семантическая ошибка: Попытка переопределения идентификатора Строка: 3 |

Продолжение таблицы 8.1

|  |  |
| --- | --- |
| Исходный код | Диагностическое сообщение |
| integer function func(integer a)  {return 1;}  main{  type integer i = func(1,2);} | Ошибка 308: Семантическая ошибка: Кол-во ожидаемых функцией и передаваемых параметров не совпадают Строка: 4 |
| integer function func(integer a)  {return 1;}  main{  type integer i = func("string");  } | Ошибка 309: Семантическая ошибка: Несовпадение типов передаваемых параметров Строка: 4 |
| main{type string i = "sdj;  } | Ошибка 311: Семантическая ошибка: Обнаружен символ '"'. Возможно, не закрыт строковый литерал Строка: 2 |

**Заключение**

В ходе выполнения курсовой работы был разработан транслятор для языка программирования ODU-2022. Таким образом, были выполнены основные задачи данной курсовой работы:

* Сформулирована спецификация языка ODU-2022;
* разработан лексический анализатор;
* разработана контекстно-свободная, приведённая к нормальной форме Грейбах, грамматика;
* разработан синтаксический анализатор;
* разработан семантический анализатор;
* разработан транслятор кода на язык ассемблера;
* проведено тестирование транслятора.

Окончательная версия языка ODU-2022 включает:

* 3 типа данных;
* поддержка операторов вывода;
* возможность вызова функций стандартной библиотеки;
* наличие 5 арифметических операторов для вычисления выражений;
* поддержка функций, процедур, операторов цикла и условия;
* структурированная и классифицированная система для обработки ошибок пользователя.

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

# **Список использованных источников**

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

2. Ахо, А. Теория синтаксического анализа, перевода и компиляции /А. Ахо, Дж. Ульман. – Москва : Мир, 1998. – Т. 2 : Компиляция. - 487 с.

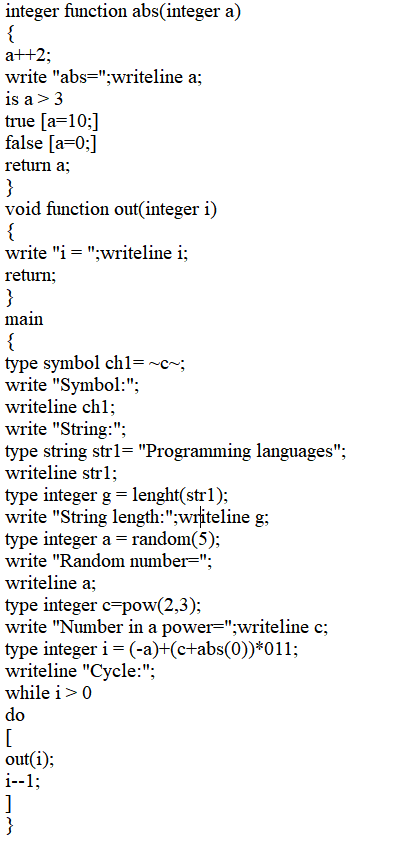
3. Герберт, Ш. Справочник программиста по C/C++ / Шилдт Герберт. - 3-е изд. – Москва : Вильямс, 2003. - 429 с.

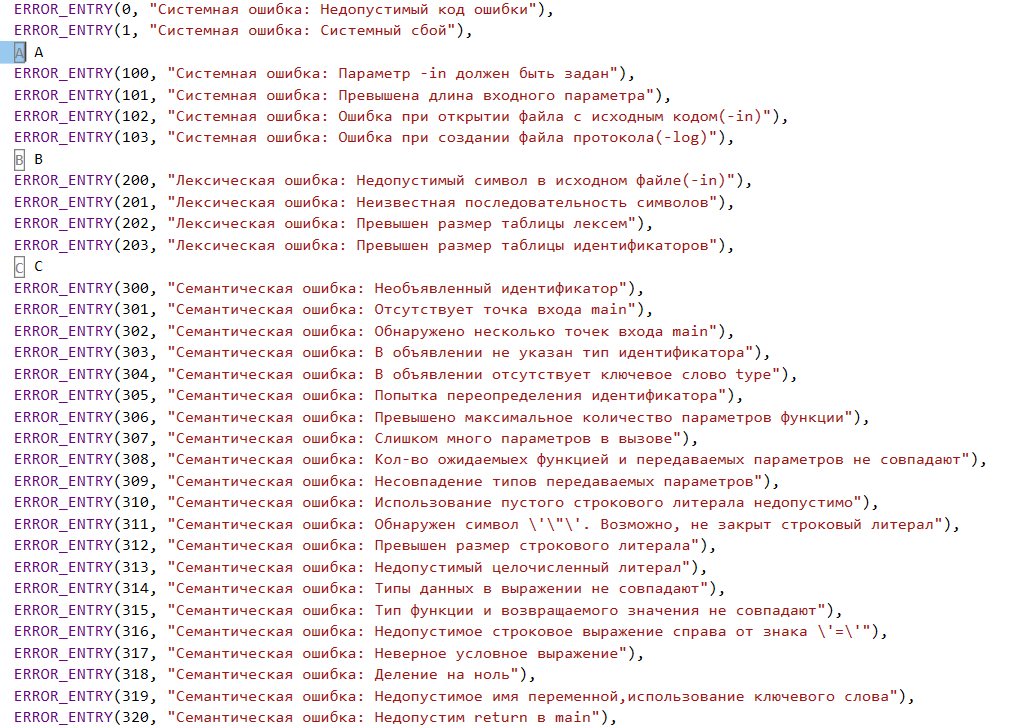
4. Прата, С. Язык программирования С++. Лекции и упражнения / С. Прата. – М., 2006 — 1104 c.

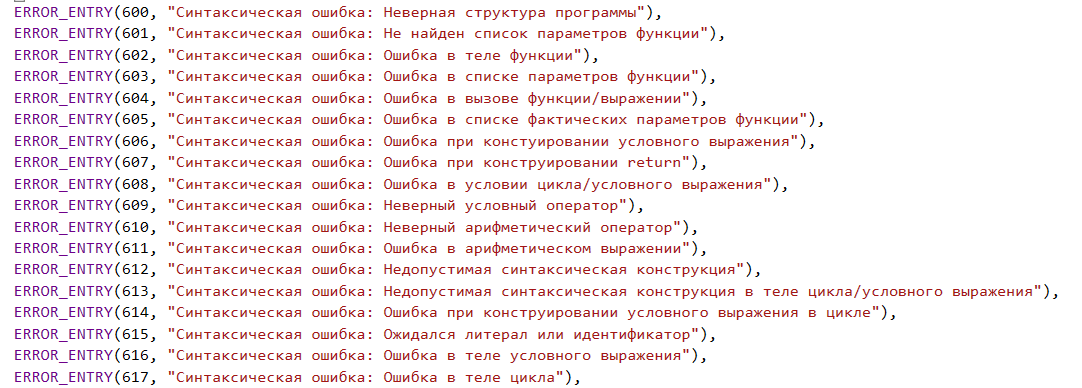
5. Страуструп, Б. Принципы и практика использования C++ / Б. Страуструп – 2009 – 1238 с

# **Приложение А**

Исходный код программы на языке ODU-2022

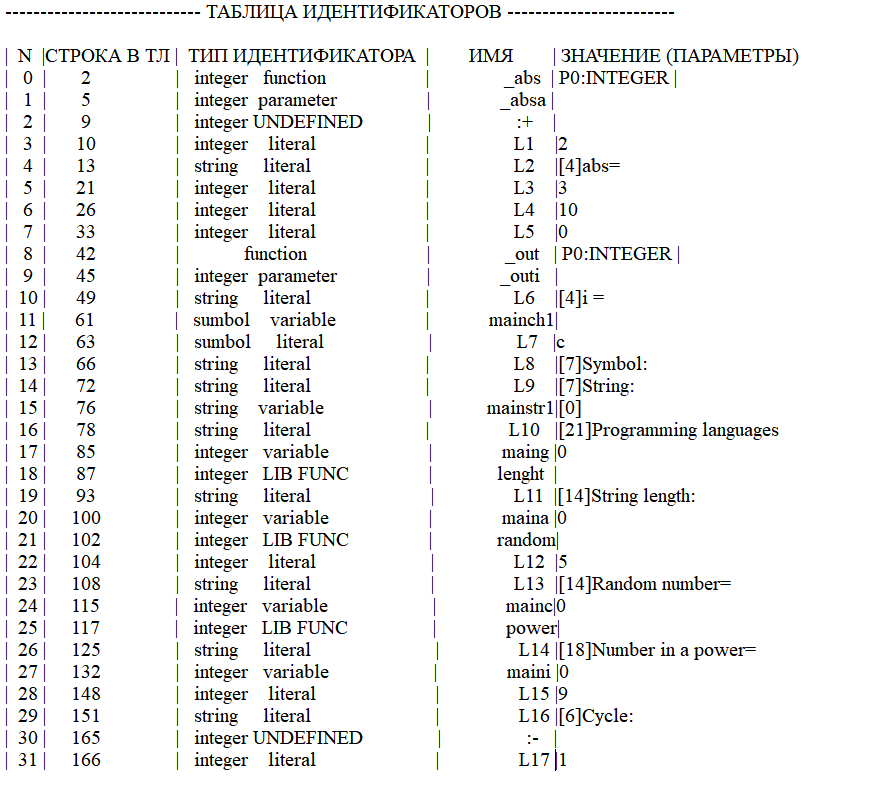


Таблица ошибок языка ODU-2022

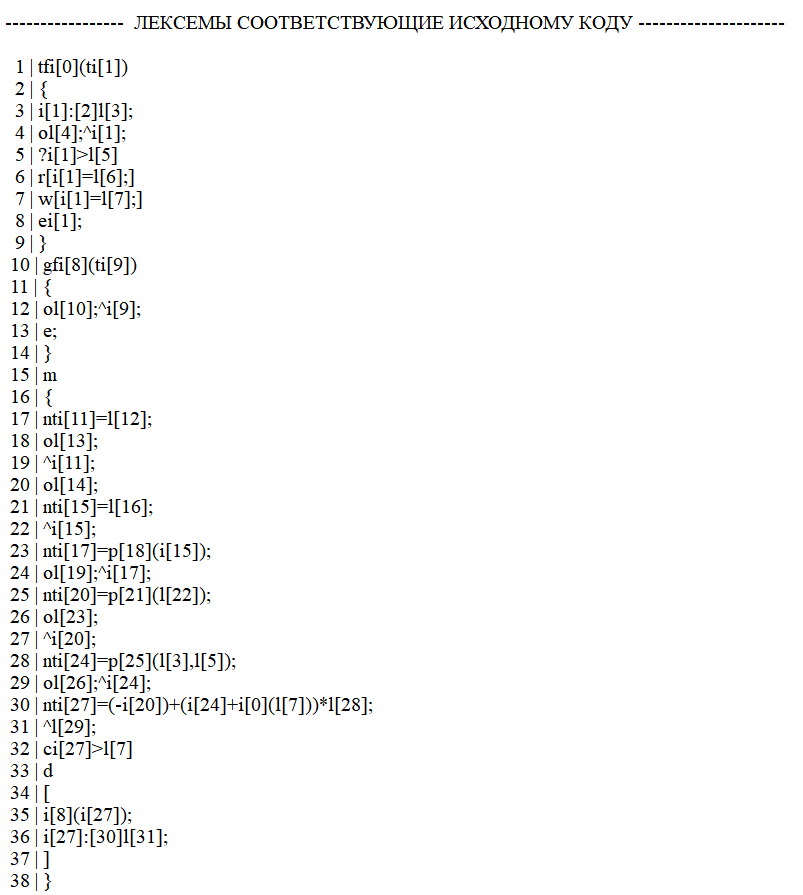


# **Приложение Б**

Таблица идентификаторов контрольного примера



Лексемы соответствующие контрольному примеру



# **Приложение В**

Грамматика языка

Greibach greibach(NS('S'), TS('$'), 19,

Rule(NS('S'), GRB\_ERROR\_SERIES, 3, // Неверная структура программы

Rule::Chain(6, TS('t'), TS('f'), TS('i'), NS('P'), NS('T'), NS('S')),

Rule::Chain(6, TS('g'), TS('f'), TS('i'), NS('P'), NS('G'), NS('S')),

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

),

Rule(NS('T'), GRB\_ERROR\_SERIES + 2, 2, // Ошибка в теле функции

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

Rule::Chain(4, TS('{'), NS('K'), NS('Q'), TS('}'))

),

Rule(NS('P'), GRB\_ERROR\_SERIES + 1, 2, // Не найден список параметров функции

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

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

),

Rule(NS('E'), GRB\_ERROR\_SERIES + 3, 2, // Ошибка в списке параметров функции

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

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

),

Rule(NS('F'), GRB\_ERROR\_SERIES + 4, 2, // Ошибка в вызове функции

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

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

),

Rule(NS('N'), GRB\_ERROR\_SERIES + 5, 4, // Ошибка в списке параметров функции

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

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

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

Rule::Chain(3, TS('l'), TS(','), NS('N'))

),

Rule(NS('R'), GRB\_ERROR\_SERIES + 6, 4, // Ошибка при констуировании условного выражения

Rule::Chain(2, TS('r'), NS('Y')),

Rule::Chain(2, TS('w'), NS('Y')),

Rule::Chain(4, TS('r'), NS('Y'), TS('w'), NS('Y')),

Rule::Chain(4, TS('w'), NS('Y'), TS('r'), NS('Y'))

),

Rule(NS('Z'), GRB\_ERROR\_SERIES + 8, 4, // Ошибка в условии цикла/условного выражения

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

Rule::Chain(3, TS('i'), NS('L'), TS('l')),

Rule::Chain(3, TS('l'), NS('L'), TS('l')),

Rule::Chain(3, TS('l'), NS('L'), TS('i'))

),

Rule(NS('L'), GRB\_ERROR\_SERIES + 9, 6,

Rule::Chain(1, TS('0')), // ==

Rule::Chain(1, TS('1')), // !=

Rule::Chain(1, TS('2')), // >

Rule::Chain(1, TS('3')), // <

Rule::Chain(1, TS('4')), // >=

Rule::Chain(1, TS('5')) // <=

),

Rule(NS('A'), GRB\_ERROR\_SERIES + 10, 5,

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

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

Rule::Chain(1, TS('\*')),

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

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

),

Rule(NS('W'), GRB\_ERROR\_SERIES + 11, 12,

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

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

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

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

Rule::Chain(6, TS('('), TS('-'), TS('i'), TS(')'), NS('A'), NS('W')),

Rule::Chain(5, TS('('), NS('W'), TS(')'), NS('A'), NS('W')),

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

Rule::Chain(2, TS('p'), NS('F')),

Rule::Chain(3, TS('i'), NS('A'), NS('W')),

Rule::Chain(3, TS('l'), NS('A'), NS('W')),

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

Rule::Chain(4, TS('p'), NS('F'), NS('A'), NS('W'))

),

Rule(NS('K'), GRB\_ERROR\_SERIES + 12, 22,

Rule::Chain(4, TS('?'), NS('Z'), NS('R'), NS('K')),

Rule::Chain(5, TS('c'), NS('Z'), TS('d'), NS('H'), NS('K')),

Rule::Chain(3, TS('?'), NS('Z'), NS('R')),

Rule::Chain(4, TS('c'), NS('Z'), TS('d'), NS('H')),

Rule::Chain(7, TS('n'), TS('t'), TS('i'), TS('='), NS('V'), TS(';'), NS('K')),

Rule::Chain(7, TS('n'), TS('t'), TS('i'), TS('='), NS('W'), TS(';'), NS('K')),

Rule::Chain(5, TS('n'), TS('t'), TS('i'), TS(';'), NS('K')),

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

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

Rule::Chain(4, TS('o'), NS('V'), TS(';'), NS('K')),

Rule::Chain(4, TS('^'), NS('V'), TS(';'), NS('K')),

Rule::Chain(6, TS('i'), TS('='), TS('U'), NS('F'), TS(';'), NS('K')),

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

Rule::Chain(6, TS('n'), TS('t'), TS('i'), TS('='), NS('V'), TS(';')),

Rule::Chain(6, TS('n'), TS('t'), TS('i'), TS('='), NS('W'), TS(';')),

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

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

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

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

Rule::Chain(3, TS('^'), NS('V'), TS(';')),

Rule::Chain(5, TS('i'), TS('='), TS('U'), NS('F'), TS(';')),

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

),

Rule(NS('X'), GRB\_ERROR\_SERIES + 13, 16,

Rule::Chain(4, TS('?'), NS('Z'), NS('R'), NS('X')),

Rule::Chain(5, TS('c'), NS('Z'), TS('d'), NS('H'), NS('X')),

Rule::Chain(3, TS('?'), NS('Z'), NS('R')),

Rule::Chain(4, TS('c'), NS('Z'), TS('d'), NS('H')),

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

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

Rule::Chain(4, TS('o'), NS('V'), TS(';'), NS('X')),

Rule::Chain(4, TS('^'), NS('V'), TS(';'), NS('X')),

Rule::Chain(6, TS('i'), TS('='), TS('U'), NS('F'), TS(';'), NS('K')),

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

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

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

Rule::Chain(3, TS('^'), NS('V'), TS(';')),

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

Rule::Chain(5, TS('i'), TS('='), TS('U'), NS('F'), TS(';')),

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

),

Rule(NS('Q'), GRB\_ERROR\_SERIES + 7, 4,

Rule::Chain(3, TS('e'), NS('V'), TS(';')),

Rule::Chain(6, TS('e'), TS('('), TS('-'), TS('i'), TS(')'), TS(';')),

Rule::Chain(2, TS('e'), TS(';')),

Rule::Chain(5, TS('e'), TS('('), TS('l'), TS(')'), TS(';'))

),

Rule(NS('Y'), GRB\_ERROR\_SERIES + 16, 4,

Rule::Chain(4, TS('['), NS('X'), NS('Q'), TS(']')),

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

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

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

),

Rule(NS('H'), GRB\_ERROR\_SERIES + 17, 4,

Rule::Chain(4, TS('['), NS('X'), NS('Q'), TS(']')),

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

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

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

),

Rule(NS('V'), GRB\_ERROR\_SERIES + 15, 2,

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

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

),

Rule(NS('U'), GRB\_ERROR\_SERIES + 3, 2,

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

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

),

Rule(NS('G'), GRB\_ERROR\_SERIES + 2, 2,

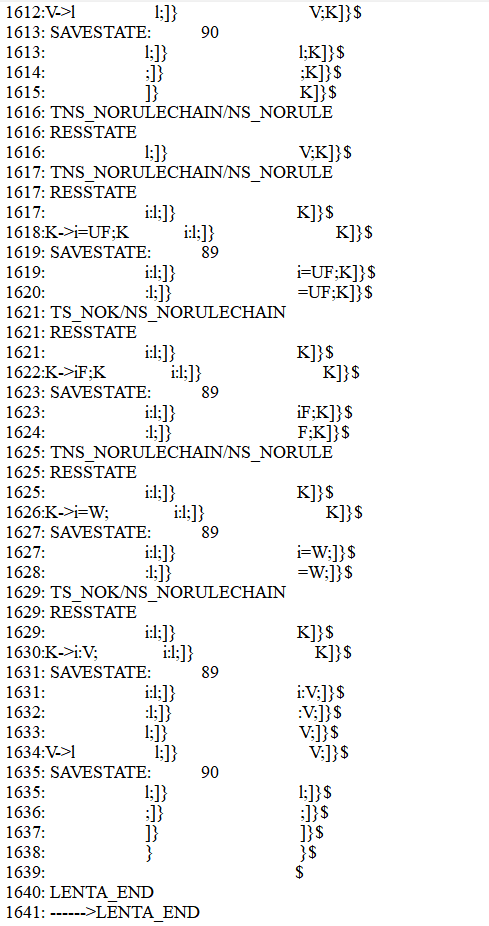
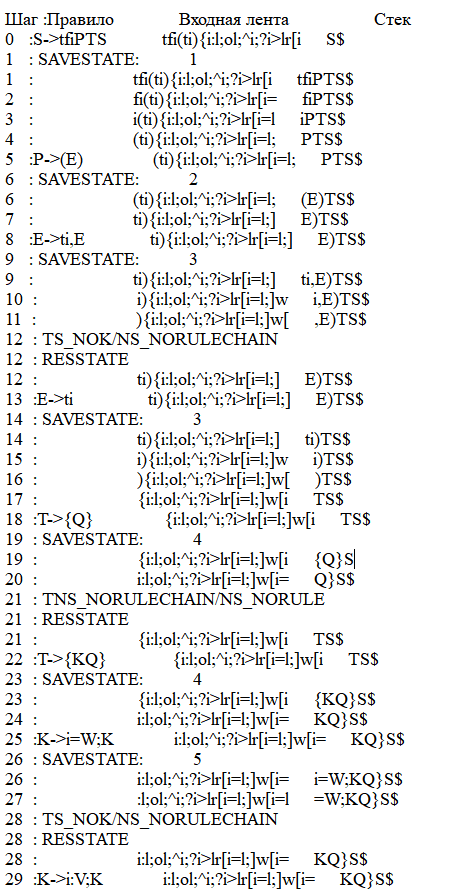
Rule::Chain(4, TS('{'), TS('e'), TS(';'), TS('}')),

Rule::Chain(5, TS('{'), NS('K'), TS('e'), TS(';'), TS('}'))

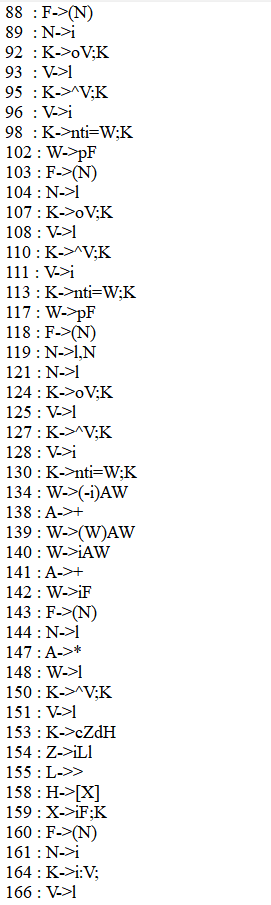
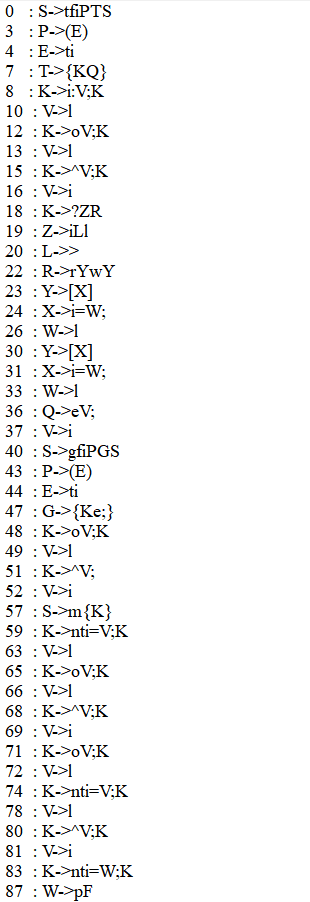
)

);

Разбор исходного кода синтаксическим анализатором

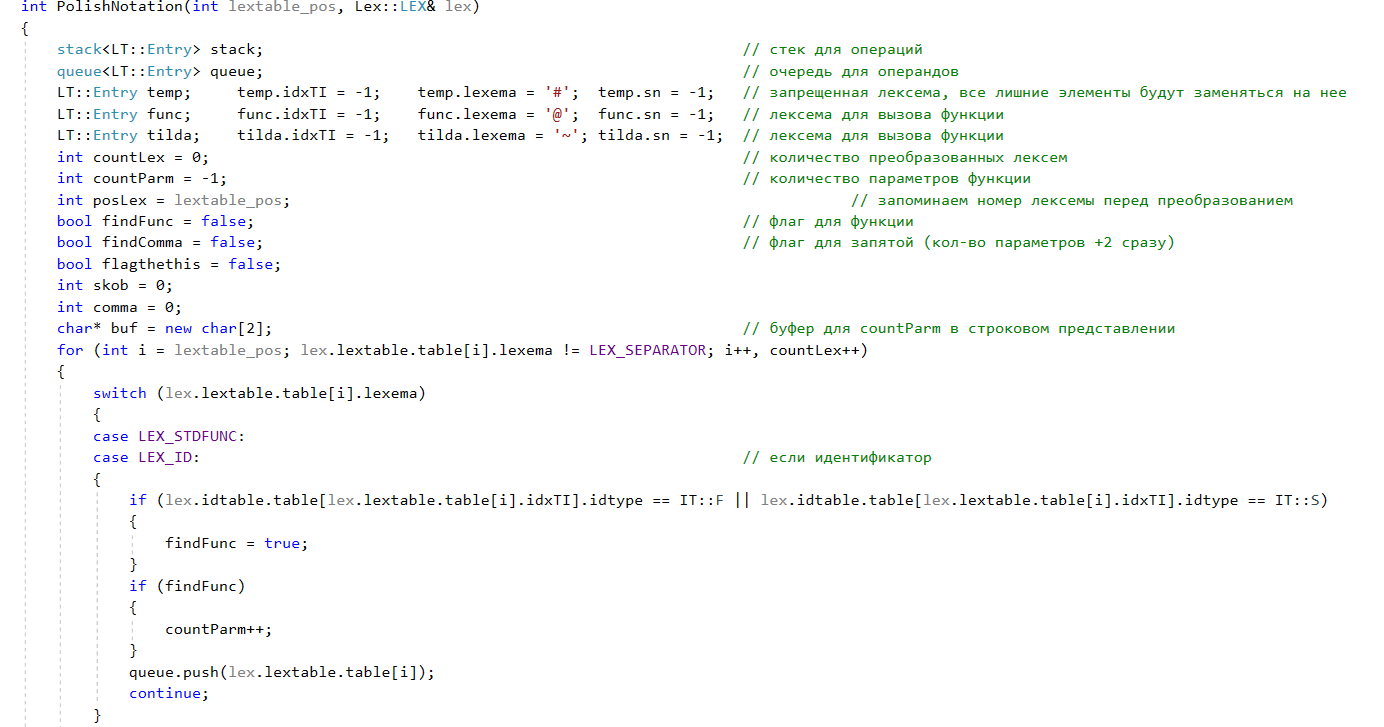


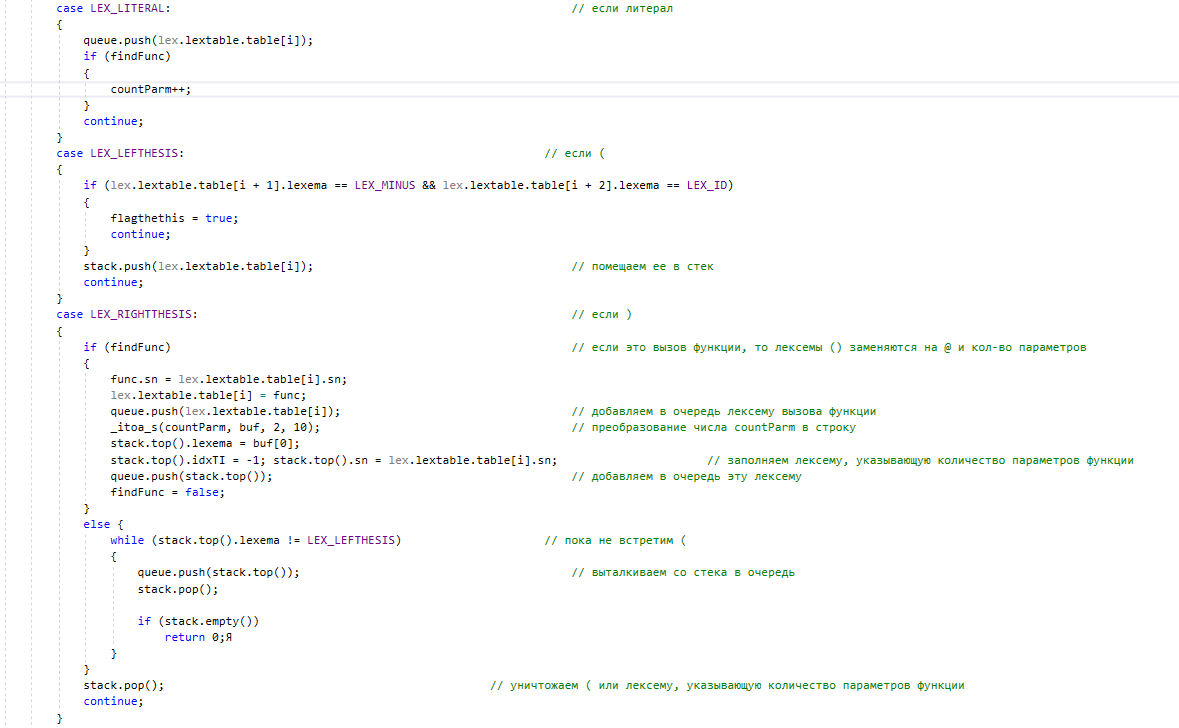
Дерево разбора

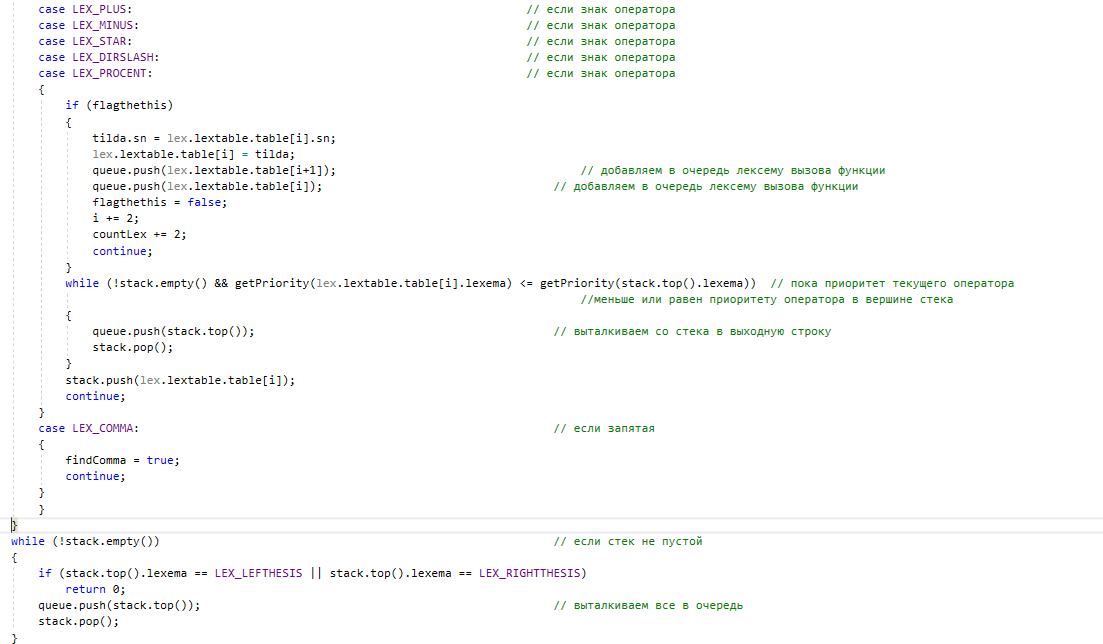


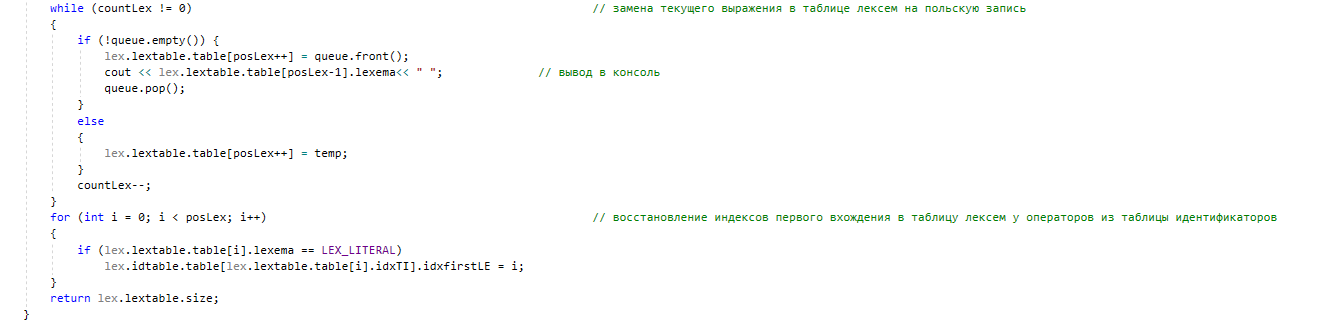
# **Приложение Г**

Программная реализация механизма преобразования в ПОЛИЗ

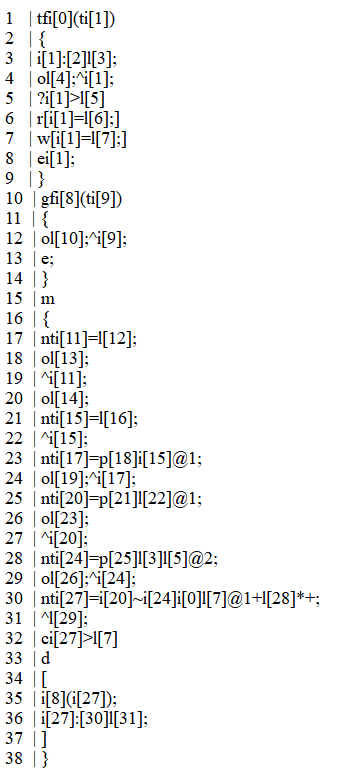








Лексемы соответствующие контрольному примеру после преобразования выражений в ПОЛИЗ



# 

# **Приложение Д**

Результат генерации кода контрольного примера в Ассемблер

