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 PVG-2022»

Выполнил студент Подобед Владислав Георгиевич

(Ф.И.О.)

Руководитель проекта асс. Мущук Артур Николаевич

(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 асс. Мущук Артур Николаевич

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

Нормоконтролер асс. Мущук Артур Николаевич

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

Курсовой проект защищен с оценкой

Минск 2022

Содержание

[Введение 4](#_Toc122602342)

[1 Спецификация языка программирования 5](#_Toc122602343)

[1.1 Характеристика языка программирования 5](#_Toc122602344)

[1.2 Определение алфавита языка программирования 5](#_Toc122602345)

[1.3 Применяемые сепараторы 6](#_Toc122602346)

[1.4 Применяемые кодировки 6](#_Toc122602347)

[1.5 Типы данных 6](#_Toc122602348)

[1.6 Преобразование типов данных 7](#_Toc122602349)

[1.7 Идентификаторы 7](#_Toc122602350)

[1.8 Литералы 7](#_Toc122602351)

[1.9 Объявление данных 8](#_Toc122602352)

[1.10 Инициализация данных 9](#_Toc122602353)

[1.11 Инструкции языка 9](#_Toc122602354)

[1.12 Операции языка 10](#_Toc122602355)

[1.13 Выражения и их вычисление 10](#_Toc122602356)

[1.14 Конструкции языка 10](#_Toc122602357)

[1.15 Области видимости идентификаторов 11](#_Toc122602358)

[1.16 Семантические проверки 11](#_Toc122602359)

[1.17 Распределение оперативной памяти на этапе выполнения 12](#_Toc122602360)

[1.18 Стандартная библиотека и её состав 12](#_Toc122602361)

[1.19 Ввод и вывод данных 13](#_Toc122602362)

[1.20 Точка входа 13](#_Toc122602363)

[1.21 Препроцессор 13](#_Toc122602364)

[1.22 Соглашения о вызовах 13](#_Toc122602365)

[1.23 Объектный код 13](#_Toc122602366)

[1.24 Классификация сообщений транслятора 13](#_Toc122602367)

[1.25 Контрольный пример 14](#_Toc122602368)

[2 Структура транслятора 15](#_Toc122602369)

[2.1 Компоненты транслятора, их назначение и принципы взаимодействия 15](#_Toc122602370)

[2.2 Перечень входных параметров транслятора 16](#_Toc122602371)

[2.3 Протоколы, формируемые транслятором 17](#_Toc122602372)

[3 Разработка лексического анализатора 18](#_Toc122602373)

[3.1 Структура лексического анализатора 18](#_Toc122602374)

[3.2. Контроль входных символов 18](#_Toc122602375)

[3.3 Удаление избыточных символов 18](#_Toc122602376)

[3.4 Перечень ключевых слов 19](#_Toc122602377)

[3.5 Основные структуры данных 20](#_Toc122602378)

[3.6 Структура и перечень сообщений лексического анализатора 20](#_Toc122602379)

[3.7 Принцип обработки ошибок 21](#_Toc122602380)

[3.8 Параметры лексического анализатора 21](#_Toc122602381)

[3.9 Алгоритм лексического анализа 21](#_Toc122602382)

[3.10 Контрольный пример 22](#_Toc122602383)

[4. Разработка синтаксического анализатора 23](#_Toc122602384)

[4.1 Структура синтаксического анализатора 23](#_Toc122602385)

[4.2 Контекстно-свободная грамматика, описывающая синтаксис языка 23](#_Toc122602386)

[4.3 Построение конечного магазинного автомата 25](#_Toc122602387)

[4.4 Основные структуры данных 26](#_Toc122602388)

[4.5 Описание алгоритма синтаксического разбора 26](#_Toc122602389)

[4.6 Структура и перечень сообщений синтаксического анализатора 26](#_Toc122602390)

[4.7. Параметры синтаксического анализатора и режимы его работы 27](#_Toc122602391)

[4.8. Принцип обработки ошибок 27](#_Toc122602392)

[4.9. Контрольный пример 27](#_Toc122602393)

[5 Разработка семантического анализатора 28](#_Toc122602394)

[5.1 Структура семантического анализатора 28](#_Toc122602395)

[5.2 Функции семантического анализатора 28](#_Toc122602396)

[5.3 Структура и перечень сообщений семантического анализатора 28](#_Toc122602397)

[5.4 Принцип обработки ошибок 29](#_Toc122602398)

[5.5 Контрольный пример 29](#_Toc122602399)

[6. Вычисление выражений 30](#_Toc122602400)

[6.1 Выражения, допускаемые языком 30](#_Toc122602401)

[6.2 Польская запись и принцип её построения 30](#_Toc122602402)

[6.3 Программная реализация обработки выражений 31](#_Toc122602403)

[6.4 Контрольный пример 31](#_Toc122602404)

[7. Генерация кода 32](#_Toc122602405)

[7.1 Структура генератора кода 32](#_Toc122602406)

[7.2 Представление типов данных в оперативной памяти 32](#_Toc122602407)

[7.3 Статическая библиотека 33](#_Toc122602408)

[7.4 Особенности алгоритма генерации кода 33](#_Toc122602409)

[7.5 Входные параметры генератора кода 34](#_Toc122602410)

[7.6 Контрольный пример 34](#_Toc122602411)

[8. Тестирование транслятора 35](#_Toc122602412)

[8.1 Тестирование проверки на допустимость символов 35](#_Toc122602413)

[8.2 Тестирование лексического анализатора 35](#_Toc122602414)

[8.3 Тестирование синтаксического анализатора 35](#_Toc122602415)

[8.4 Тестирование семантического анализатора 36](#_Toc122602416)

[Заключение 38](#_Toc122602417)

[Список использованных источников 39](#_Toc122602418)

[Приложение А 40](#_Toc122602419)

[Приложение Б 44](#_Toc122602420)

[Приложение В 46](#_Toc122602421)

[Приложение Г 49](#_Toc122602422)

[Приложение Д 51](#_Toc122602423)

# Введение

Задачей данного курсового проекта была поставлена разработка транслятора своего языка программирования PVG-2022. Этот язык программирования предназначен для выполнения простейших операций и арифметических действий над числами.

Главной задачей транслятора заключается в том, чтобы сделать исходный код на данном языке программирования понятной компьютеру. Для решения этой задачи был выбран способ трансляции исходного кода моего языка программирования в исходный код на языке ассемблера.

Исходя из цели курсового проекта, были определены следующие задачи:

– разработка спецификации языка программирования;

– разработка структуры транслятора;

– разработка лексического анализатора;

– разработка синтаксического анализатора;

– разработка семантического анализатора;

– обработка выражений с помощью польской инверсии;

– генерация кода на язык ассемблера;

– тестирование транслятора;

Способы решения каждой задачи будут описаны в соответствующих главах курсового проекта.

В первой главе работы определена спецификация языка программирования.

Во второй главе представлена структура транслятора. В ней перечислены компоненты транслятора, их назначения и принципы взаимодействия.

В третьей главе описана разработка лексического анализатора, который создаёт таблицы лексем и идентификаторов.

В четвертой главе описана разработка синтаксического анализатора, который выполняет разбор исходного кода в соответствии с правилами языка программирования.

В пятой главе описан семантический анализатор, которые проверяет исходный код программы на наличие семантических ошибок.

В шестой главе описан способ преобразования выражений в польский формат.

В седьмой главе представлена генерация кода в язык ассемблера с помощью таблиц лексем и идентификаторов

В восьмой главе описывается тестирование транслятора

# 1 Спецификация языка программирования

## 1.1 Характеристика языка программирования

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

## 1.2 Определение алфавита языка программирования

Алфавит языка программирования – набор символов, которые могут использоваться при написании исходного кода.

Алфавит языка PVG-2022 основан на кодировке Windows-1251, представленной на рисунке 1.1.



Рисунок 1.1 – Алфавит входных символов

На этапе выполнения могут использоваться символы латинского алфавита, цифры десятичной системы счисления от 0 до 9, спецсимволы, а также непечатные символы пробела, табуляции и перевода строки. Русские символы разрешены только в строковых литералах.

## 1.3 Применяемые сепараторы

Символы-сепараторы служат в качестве разделителей цепочек языка во время обработки исходного текста программы с целью разделения на токены. Они представлены в таблице 1.1.

Таблица 1.1 – Символы-сепараторы

|  |  |  |
| --- | --- | --- |
| Сепаратор | Название | Область применения |
| ‘ ‘ | Пробел | Допускается везде, кроме идентификаторов и ключевых слов |
| ; | Точка с запятой | Разделение конструкций |
| {…} | Фигурные скобки | Заключение программного блока |
| […] | Квадратные кавычки | Блок кода |
| (…) | Круглые скобки | Приоритет операций, параметры функции |
| “…” | Двойные кавычки | Строковый литерал |
| ‘…’ | Одинарные кавычки | Допускается везде, кроме идентификаторов и ключевых слов |
| = | Знак «равно» | Присваивание значения |
| , | Запятая | Разделение параметров |
| /  \  +  -  \*  : | Знаки «косая черта», «обратная косая черта», «плюс», «астерикс», «тильда», | Выражения |
| &  ^  <  > | «амперсанд», «циркумфлекс», знаки «больше» и «меньше» | Выражения в операторе цикла |

## 1.4 Применяемые кодировки

Для написания исходного кода на языке программирования PVG-2022 используется кодировка Windows-1251.

## 1.5 Типы данных

В языке PVG-2022 реализованы четыре типа данных: целочисленный беззнаковый(uint), символьный(char), строковый(str), логический(bool). Описание типов данных, предусмотренных в данным языке представлено в таблице 1.2.

Таблица 1.2 – Типы данных языка PVG-2022

|  |  |
| --- | --- |
| Тип данных | Описание типа данных |
| Целочисленный тип данных uint | Фундаментальный тип данных. Используется для работы с целочисленными положительными значениями. В памяти занимает 1 байт.  При попытке инициализации значением больше максимального, инициализируется максимальным.  Максимальное значение: 255. Минимальное значение: 0.  Инициализация по умолчанию: значение 0. |
| Строковый тип данных str | Фундаментальный тип данных. Используется для работы с набором символов, каждый символ в памяти занимает 1 байт. Инициализация по умолчанию: символ конца строки “\0”. Для типа str определён оператор + – конкатенация строк. |
| Логический тип данных bool | примитивный тип данных в информатике, принимающий два возможных значения, иногда называемых истиной (true) и ложью (false). В некоторых языках программирования за значение истина полагается 1, за значение ложь — 0. |
| Символьный тип данных char | Фундаментальный тип данных. Используется для работы с символом, который в памяти занимает 1 байт. Инициализация по умолчанию: символ конца строки “\0”. |

## 1.6 Преобразование типов данных

Преобразование типов данных не поддерживается, т.е. язык является строготипизированным. Но в стандартной библиотеке есть функции преобразования типа uint в тип char и наоборот.

## 1.7 Идентификаторы

В имени идентификатора допускаются только символы латинского алфавита и знак «\_». Максимальная длина имени идентификатора – 8 символов. Максимальная длина имени идентификатора функции – 8 символов. При вводе идентификатора длиной более разрешенного количества символов, он будет усекаться. Имя идентификатора не может совпадать с именем функции, уже содержащаяся в стандартной библиотеке, если только это функция подключена через оператор export.

## 1.8 Литералы

С помощью литералов осуществляется инициализация переменных. В языке существует два типа литералов. Краткое описание литералов языка PVG-2022 представлено в таблице 1.3.

Таблица 1.3 – Описание литералов

|  |  |  |  |
| --- | --- | --- | --- |
| Тип литерала | Регулярное выражение | Пояснение | Пример |
| Целочисленный литерал | [1-9]+[0-9]\* | Целочисленные неотрицательные литералы, по умолчанию инициализируются 0. Литералы могут быть только rvalue. | let uint sum = 15;  15 – целочисленный литерал. |
| Символьный литерал | [a-z|A-Z|А-Я|а-я|0-9|!-/] | Символ, заключённый в ‘’ (одинарные кавычки), по умолчанию инициализируются пустой строкой. Литералы могут быть только rvalue. | let char symbol = ‘T’;  T – символьный литерал. |
| Строковый литерал | [a-z|A-Z|А-Я|а-я|0-9|!-/]\* | Строковые литералы, максимальная длина строки 255 символов | output “aaa” |

Литералы являются константами и при генерации кода объявляются один раз.

## 1.9 Объявление данных

Для объявления переменной используется ключевое слово **let**, потом со следующей строки указывается тип данных и имя идентификатора.

Пример объявления числового типа с инициализацией:

let uintdemo;

demo = 32;

Пример объявления переменной строкового типа с инициализацией:

let stringbl;

bl = "vlad";

Пример объявления переменной символьного типа с инициализацией:

let charstr1;

str1 = 'q';

Для объявления функций используется ключевое слово function. Далее обязателен список параметров и тело функции.

Область видимости «сверху вниз» (по принципу С++). В языке PVG-2022 требуется обязательное объявление переменной перед её инициализацией и последующим использованием. Все переменные должны находиться внутри программного блока. Имеется возможность объявления одинаковых переменных в разных блоках, т. к. переменные, объявленные в одной функции, недоступны в другой. Каждая переменная получает префикс – название функции, в которой она объявлена. Объявление функций стандартной библиотеки можно производить в любом месте кода.

## 1.10 Инициализация данных

При объявлении переменной не допускается инициализация. Описание способов инициализации переменных языка PVG-2022 представлено в таблице 1.4.

Таблица 1.4 – Способы инициализации переменных

|  |  |  |
| --- | --- | --- |
| Конструкция | Описание | Пример |
| let <тип данных> <идентификатор>; | Автоматическая инициализация: переменные типа uint инициализируются нулём, переменные типа char – пустым символом. | let uint sum;  let symbol chr; |
| <идентификатор> = <значение>; | Присваивание переменной значения. | sum = 15;  chr = ‘S’; |

Соответствие типов проверяется на синтаксическом анализе.

## 1.11 Инструкции языка

Инструкции языка PVG-2022 представлены в таблице 1.5.

Таблица 1.5 – ИнструкцииязыкаPVG-2022

|  |  |
| --- | --- |
| Инструкция | Реализация |
| Объявление переменной | let <тип данных> <идентификатор>; |
| Объявление функции | let <тип данных> func <идентификатор> (<тип данных> <идентификатор>, …) {<блок кода>}; |
| Присваивание | <идентификатор> = <значение>/<идентификатор>; |
| Объявление внешней функции | export <тип данных> func <идентификатор> (<тип данных> <идентификатор>, …); |
| Блок инструкций | main  {  …  } |
| Возврат из подпрограммы | return <идентификатор> / <литерал>; |
| Условная инструкция | while(<условие>)[<блок кода>]; |
| Вывод данных | output <идентификатор> / <литерал>; |
| Однострочный комментарий до конца строки | #<любой текст> |

## 1.12 Операции языка

В языке PVG-2022 может выполнять операции, представленные в таблице 1.6. Операция сдвига учитывает только первый младший бит оператора, т.к. сдвиг более чем на 255 любого числа кроме нуля вернет число большее, чем можно разместить в типе данных uint.

Таблица 1.6 – Операции языка PVG-2022

|  |  |  |  |
| --- | --- | --- | --- |
| Операция | Примечание | Типы данных | Пример |
| ( | Приоритет операций | - | sum = (a + b) \* c; |
| ) |
| + | Суммирование | (uint, uint)  (char, char) | sum = a + b; |
| - | Вычитание | (uint, uint) | diff = a – b; |
| \* | Умножение | (uint, uint) | mul = a\*b; |
| : | Деление | (uint, uint) | div = a:b; |
| % | Остаток от деления | (uint, uint) | mod = a%b; |
| / | Сдвиг влево | (uint, uint) | pr = a / b; |
| \ | Сдвиг вправо | (uint, uint) | pr = a \ b; |
| = | Присваивание | (uint, uint)  (char, char)  (str, str) | sum = 15;  chr = ‘T’; |
| <,> | Знаки «больше», «меньше» для условной инструкции | (uint, uint)  (char, char) | while(sum < diff) […]; |
| & | Оператор эквивалентности | (uint, uint)  (char, char) | while(sum & diff) […]; |
| ^ | Оператор неравенства | (uint, uint)  (char, char) | while(sum ^ diff) […]; |

Т.к. отрицательные числа не поддерживаются, если результат операции меньше нуля, он вычитается из максимального значения.

## 1.13 Выражения и их вычисление

Круглые скобки в выражении используются для изменения приоритета операций. Также не допускается запись двух подряд идущих арифметических операций. Выражение может содержать вызов функции, если эта функция уже содержится в стандартной библиотеке. Выражения вычисляются только после оператора присваивания.

## 1.14 Конструкции языка

Ключевые программные конструкции языка программирования PVG-2022 представлены в таблице 1.7.

Таблица 1.7 – Программные конструкции языка PVG-2022

|  |  |
| --- | --- |
| Конструкция | Запись на языке PVG-2022 |
| Главная функция (точка входа) | main  {  …  return <идентификатор> / <литерал>;  } |
| Функция | let <тип данных> func <идентификатор> (<тип> <идентификатор>, …)  {… return <идентификатор> / <литерал>; }; |
| Цикл | while(a^8)[ …]; |

Принцип действия while схож с одноименным циклом в C++;

## 1.15 Области видимости идентификаторов

В языке PVG-2022 все переменные являются локальными, т.е. имеют функциональную область видимости. Они обязаны находится внутри программного блока функций (по принципу С++). Объявление глобальных переменных не предусмотрено. Объявление пользовательских областей видимости не предусмотрено.

## 1.16 Семантические проверки

В языке программирования PVG-2022 выполняются следующие семантические проверки:

1. Идентификаторы не должны повторно объявляться в пределах одной функции;
2. Тип возвращаемого значения должен совпадать с типом функции при её объявлении или подключении;
3. Тип данных передаваемых значений в функцию должен совпадать с типом параметров при её объявлении или подключении;
4. В функцию должно быть передано то число параметров, сколько ожидается;
5. Тип данных результата выражения должен совпадать с типом данных идентификатора, которому оно присваивается;
6. Типы данных операндов выражения должны быть одинаковыми;
7. Для типа данных str доступна только операция сложения (конкатенация срок);
8. Тип данных str не может быть аргументом условной конструкции;
9. Для типа char определены только операции + и -;
10. Функции не должны подключаться дважды в пределах одной программы.

## 1.17 Распределение оперативной памяти на этапе выполнения

Все переменные размещаются в куче.

## 1.18 Стандартная библиотека и её состав

Стандартная библиотека PVG-2022 написана на языке программирования C++.

Для использования функций стандартной библиотеки, нужно явно подключить необходимую функцию с помощью ключевого слова export, далее работа с ними производится как с пользовательскими функциями. Функции стандартной библиотеки с описанием представлены в таблице 1.9.

Таблица 1.9 – Состав стандартной библиотеки

|  |  |  |
| --- | --- | --- |
| Функция(C++) | Возвращаемое значение | Описание |
| int ord(char) | char | Возвращает код символа |
| char chr(int) | char | Возвращает символ с заданным кодом |
| int GetMonth(int number) | uint | Возвращает номер месяца |
| int GetDate(int number) | uint | Возвращает дату в формате ДДММГГГГ |
| int GetHours(int number) | uint | Возвращает час |
| int GetMinutes(int number) | uint | Возвращает минуты |

Так же в библиотеке присутствуют приватные функции. Их описание представлено в таблице 1.10.

Таблица 1.10 – Приватные функции стандартной библиотеки

|  |  |  |
| --- | --- | --- |
| Функция(C++) | Возвращаемое значение | Описание |
| void outputuint (unsigned int a) | – | Выводит число на экран Вызывается оператором output |
| void outputchar (char a) | – | Выводит символ на экран Вызывается оператором output |
| void outputstring (void\* in) | – | Выводит строку на экран Вызывается оператором output |

Приватные функции не могут быть вызваны явно и не требуют предварительного пользовательского подключения. Они вызываются специальными операторами языка.

## 1.19 Ввод и вывод данных

В языке PVG-2022 не реализованы средства ввода данных.

Для вывода данных в стандартный поток вывода предусмотрен оператор output, который базируется на приватных функциях стандартной библиотеки.

## 1.20 Точка входа

В языке PVG-2022 каждая программа должна содержать главную функцию main, т. е. точку входа, с которой начнется последовательное выполнение программы.

## 1.21 Препроцессор

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

## 1.22 Соглашения о вызовах

В языке вызов функций происходит по соглашению о вызовах stdcall. Особенности stdcall:

– все параметры функции передаются через стек;

– память высвобождает вызываемый код;

– занесение в стек параметров идёт справа налево.

## 1.23 Объектный код

Язык PVG-2022 транслируется в язык ассемблера.

## 1.24 Классификация сообщений транслятора

Генерируемые транслятором сообщения определяют степень его информативности, то есть сообщения транслятора должны давать максимально полную информацию о допущенной пользователем ошибке при написании программы. Сообщения транслятора приведены в таблице 1.11, а также в приложении А.

Таблица 1.11 – Классификация ошибок

|  |  |
| --- | --- |
| Номера ошибок | Характеристика |
| 0 – 99 | Системные ошибки |
| 100-109 | Ошибки параметров |
| 110-119 | Ошибки открытия и чтения файлов |
| 120-129 | Ошибки лексического анализа |
| 130-139 | Ошибки таблиц лексем и таблиц идентификаторов |
| 600-699 | Ошибки синтаксического анализа |
| 700-900 | Ошибки семантического анализа |

## 1.25 Контрольный пример

Код контрольного примера представлен в Приложении А.

# 2 Структура транслятора

## 2.1 Компоненты транслятора, их назначение и принципы взаимодействия

В языке PVG-2022 исходный код транслируется в язык Assembler. Транслятор языка разделён на отдельные части, которые взаимодействуют между собой и выполняют отведённые им функции, которые представлены в пункте 2.1. Для того чтобы получить ассемблерный код, используется выходные данные работы лексического анализатора, а именно таблица лексем и таблица идентификаторов. Для указания выходных файлов используются входные параметры транслятора, которые описаны в таблице 2.1. Структура транслятора языка PVG-2022 приведена на рисунке 1.



Рисунок 2.1 – Структура транслятора языка программирования PVG-2022

Первой фазой работы компилятора называется лексическим анализом, а программа, её реализующая, – лексическим анализатором (сканером). На вход лексического анализатора подаётся последовательность символов входного языка. Он производит предварительный разбор текста, преобразующий единый массив текстовых символов в массив отдельных слов (в теории компиляции вместо термина «слово» часто используют термин «токен»). 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. Таблица лексем (ТЛ) и таблица идентификаторов (ТИ) являются входом для следующей фазы компилятора – синтаксического анализа (разбора, парсера).

Цели лексического анализатора:

− убрать все лишние пробелы;

− выполнить распознавание лексем;

− построить таблицу лексем и таблицу идентификаторов;

− при неуспешном распознавании или обнаружении некоторых ошибок во входном тексте выдать сообщение об ошибке.

Синтаксический анализатор – 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 анализатор – часть транслятора, выполняющая семантический анализ, то есть проверку исходного кода на наличие ошибок, которые невозможно отследить при помощи регулярной и контекстно-свободной грамматики. Входными данными являются таблица лексем и идентификаторов.

Генератор кода – часть транслятора, выполняющая генерацию ассемблерного кода на основе полученных данных на предыдущих этапах трансляции. На вход генератора подаются таблица лексем и таблица идентификаторов, на основе которых генерируется файл с ассемблерным кодом.

## 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 PVG-2022

|  |  |  |
| --- | --- | --- |
| Входной параметр | Описание | Значение по умолчанию |
| -in:<имя\_файла> | Входной файл с любым расширением, в котором содержится исходный код на языке PVG-2022. Данный параметр должен быть указан обязательно. В случае если он не будет задан, то выполнение этапа трансляции не начнётся. | Не предусмотрено |
| -log:<имя\_файла> | Файл содержит в себе краткую информацию об исходном коде на языке PVG-2022. В этот файл могут быть выведены таблицы идентификаторов, лексем, а также дерево разбора. | <имя\_файла>.log |
| -out:<имя\_файла> | В этот файл будет записан результат трансляции кода на язык assembler | <имя\_файла>.asm |
| m | Вывод дерева разбора синтаксического анализатора. | – |
| l | Вывод таблицы лексем | – |

Продолжение таблицы 2.1

|  |  |  |
| --- | --- | --- |
| i | Вывод таблицы идентификаторов | – |

Таблицы лексем и дерево разбора синтаксического анализатора выводятся в лог журнал.

## 2.3 Протоколы, формируемые транслятором

Таблица с перечнем протоколов, формируемых транслятором языка PVG-2022 и их назначением представлена в таблице 2.2.

Таблица 2.2 – Протоколы, формируемые транслятором языка PVG-2022

|  |  |
| --- | --- |
| Формируемый протокол | Описание протокола |
| Файл журнала, “\*.log ” | Файл содержит в себе краткую информацию об исходном коде на языке PVG-2022. В этот файл выводится протокол работы анализаторов, а так же различные ошибки |
| “\*.asm” | Содержит сгенерированный код на языке Ассемблера. |

В log файл выводятся все ошибки, за исключением тех, что связаны с открытием файла log или считывания параметров.

# 3 Разработка лексического анализатора

## 3.1 Структура лексического анализатора

Лексический анализатор – часть транслятора, выполняющая лексический анализ. Лексический анализатор принимает обработанный и разбитый на отдельные компоненты исходный код на языке PVG-2022. На выходе формируется таблица лексем и таблица идентификаторов. Структура лексического анализатора представлена на рисунке 3.1.



Рисунок 3.1 – Структура лексического анализатора

## 3.2. Контроль входных символов

Исходный код на языке программирования PVG-2022 прежде чем транслироваться проверяется на допустимость символов. То есть изначально из входного файла считывается по одному символу и проверяется является ли он разрешённым.

Таблица для контроля входных символов представлена в приложении Б.

Принцип работы таблицы заключается в соответствии значения каждому элементу в шестнадцатеричной системе счисления значению в таблице ASCII.

Описание значения символов: T – разрешённый символ, F – запрещённый символ, S – пробельный символ, С – символ одинарной кавычки, L – символ-разделитель, D – символ двойной кавычки, O – символ начала комментария, N – символ новой строки.

## 3.3 Удаление избыточных символов

Удаление избыточных символов не предусмотрено, так как после проверки на допустимость символов исходный код на языке программирования PVG-2022 разбивается на токены, которые записываются в очередь.

.

## 3.4 Перечень ключевых слов

Лексемы – это символы, соответствующие ключевым словам, символам операций и сепараторам, необходимые для упрощения дальнейшей обработки исходного кода программы. Данное соответствие описано в таблице 3.2.

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

|  |  |  |  |
| --- | --- | --- | --- |
| Тип цепочки | Примечание | Цепочка | Лексема |
| Тип данных | Целочисленный тип данных | uint | i |
| Строковый тип данных | str | i |
| Символьный тип данных | char | i |
| Лексема | Объявление переменной | let | v |
| Подключение функции библиотеки | export | e |
| Оператор вывода | output | p |
| Объявление функции | func | f |
| Возврат значения из функции | return | r |
| Инструкция цикла | while | u |
| Блок инструкции цикла | [ | [ |
| ] | ] |
| Блок функции | { | { |
| } | } |
| Изменение приоритетности в выражении и отделение параметров функций | ( | ( |
| ) | ) |
| Сепараторы | ; | ; |
| , | , |
| Оператор присваивания | = | v |
| Условный оператор | < | b |
| > | b |
| & | b |
| ^ | b |
| Оператор | Знаки арифметических операций | + | v |
| - | v |
| \* | v |
| / | v |
| \ | v |
| : | v |
| % | v |
| Идентификатор |  | [a-z|A-Z]+  [a-z|A-Z|0-9]\* | i |

Продолжение таблицы 3.2 – Соответствие ключевых слов, символов операций и сепараторов с лексемами

|  |  |  |  |
| --- | --- | --- | --- |
| Тип цепочки | Примечание | Цепочка | Лексема |
| Литерал | Целочисленный литерал | [1-9]+[0-9]\* | l |
| Символьный литерал | [a-z|A-Z|0-9]\*  кроме ‘ | l |
| Строковый литерал | [a-z|A-Z|0-9]\* кроме ” | l |
| Точка входа |  | main | m |

Каждому выражению соответствует детерминированный конечный автомат, то есть автомат с конечным состоянием, по которому происходит разбор данного выражения. На каждый автомат в массиве подаётся фраза 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. Пример реализации таблицы лексем представлен в приложении В.

Также в приложении В находятся конечные автоматы, соответствующие лексемам языка PVG-2022.

## 3.5 Основные структуры данных

Основные структуры таблиц лексем и идентификаторов данных языка PVG-2022, используемых для хранения, представлены в приложении В. В таблице лексем содержится лексема, её номер, полученный при разборе, номер строки в исходном коде, номер столбца в исходном коде, индекс таблицы идентификаторов (если нет соответствующего идентификатора, то индекс равен -1), а также специальное поле, в котором хранится значение лексемы. В таблице идентификаторов содержится имя идентификатора, номер в таблице лексем, тип данных, тип идентификатора, его значение, а также бинарное поле для определения внешний ли идентификатор.

## 3.6 Структура и перечень сообщений лексического анализатора

Индексы ошибок, обнаруживаемых лексическим анализатором, находятся в диапазоне 120-125. Также сам текст ошибки содержит в себе префикс [LA]. Перечень сообщений лексического анализатора представлен на листинге 3.1.

|  |
| --- |
| ERROR\_ENTRY(120, "[LA]: Ошибка при разборе токена"),  ERROR\_ENTRY(121, "[LA]: Используется необъявленный идентификатор"),  ERROR\_ENTRY(122, "[LA]: Итендификатор не имеет типа"),  ERROR\_ENTRY(124, "[LA]: Отсутствует точка входа"),  ERROR\_ENTRY(125, "[LA]: Обнаружена вторая точка входа"), |

Листинг 3.1 – Сообщения лексического анализатора

## 3.7 Принцип обработки ошибок

Все ошибки являются критическими и приводят к прекращению работы транслятора и выводу диагностического сообщения в лог файл.

## 3.8 Параметры лексического анализатора

Входные параметры используются для вывода результата работы лексического анализатора. Они передаются аргументами через командную строку и рассмотрены в таблице 2.1

## 3.9 Алгоритм лексического анализа

Алгоритм работы лексического анализа заключается в последовательном распознавании и разборе цепочек исходного кода и заполнение таблиц идентификаторов и лексем. Лексический анализатор производит распознаёт и разбирает цепочки исходного текста программы. Это основывается на работе конечных автоматов, которую можно представить в виде графов. В случае, если подходящий автомат не был обнаружен, запоминается номер строки, в которой находился этот токен и выводится сообщение об ошибке. Если токен разобран, то дальнейшие действия, которые будут с ним производиться, будут зависеть от того, чем он является. Регулярные выражения — аналитический или формульный способ задания регулярных языков. Они состоят из констант и операторов, которые определяют множества строк и множество операций над ними. Любое регулярное выражение можно представить в виде графа.

В случае, если токен является идентификатором, перед его именем записывается название функции в которой он объявлен и после этого он заносится в таблицу идентификаторов.

В случае, если токен является идентификатором функции, название функции в которой он объявлен не записывается.

В случае, если токен является литералом, то он заносится в таблицу идентификаторов в виде abi, где a – имя функции, где объявлен литерал, b – “$LEX”, c –количество определённых литералов+1.

Когда встречаем токен, являющийся ключевым словом, которое отвечает за тип данных или вид идентификатора, заносим лексему, соответствующую ему, в таблицу лексем и запоминаем тип данных или вид идентификатора, которому он соответствует. В последствии, когда встречаем идентификатор, заносим его в таблицу идентификаторов с соответствующим ему типом данных и видом идентификатора, и именем вида “ab”, где a – имя функции, где объявлен идентификатор, b – имя идентификатора.

Пример. Регулярное выражение для ключевого слова let: «let».

Граф конечного автомата для этой лексемы представлен на рисунке 3.4. S0 – начальное состояние, S3  – конечное состояние автомата.

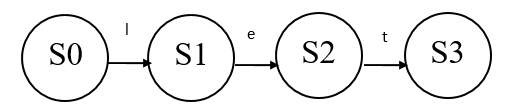


Рисунок 3.3 – Граф переходов для цепочки «let»

## 3.10 Контрольный пример

Результат работы лексического анализатора – таблицы лексем и идентификаторов – 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 PVG-2022 используется контекстно-свободная грамматика типа II в иерархии Хомского (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

|  |  |  |
| --- | --- | --- |
| Нетерминал | Цепочки правил | Описание |
| S | m{NrE;};  m{rE;};  dtfi(F){NrE;};S  dtfi(F){rE;};S  dtfi(){NrE;};S  dtfi(){rE;};S  m{rE;};S | Порождает правила, описывающее общую структуру программы |
| N | dti;  rE;  ivE;  etfi(F);  dti;N  ivE;N  etfi(F);N  pi;  pl;  pi;N  pl;N  u(B)[N];  u(B)[N];N  etfi();N  etfi(); | Порождает правила, описывающие конструкции языка |
| E | i  l  (E)  i(W)  iM  lM  (E)  i(W)  i()  i(M)  l(M)  (E)M  i(W)M  i(W) | Порождает правила, описывающие выражения |

Продолжение таблицы 4.1

|  |  |  |
| --- | --- | --- |
| Нетерминал | Цепочки правил | Описание |
| F | ti  ti,f | Порождает правила, описывающие параметры локальной функции при её объявлении |
| W | i  l  i,W  l,W | Порождает правила, описывающие принимаемые параметры функции |
| B | ibi  ibl  lbi  lbl | Порождает правила, описывающие условное выражение в операторе цикла |
| M | vE  v(E)  v(E)M  vEM | Порождает правила, описывающие знаки арифметических операций |

Протокол и ошибки работы синтаксического анализатора выводятся в лог журнал.

## 4.3 Построение конечного магазинного автомата

Конечный автомат с магазинной памятью представляет собой семерку, описание которой представлено в таблице 4.2.

Таблица 4.2 – Описание компонентов магазинного автомата

|  |  |  |
| --- | --- | --- |
| Компонента | Определение | Описание |
|  | Множество состояний автомата | Состояние автомата представляет из себя структуру, содержащую позицию на входной ленте, номера текущего правила и цепочки и стек автомата |
|  | Алфавит входных символов | Алфавит представляет из себя множества терминальных и нетерминальных символов, 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 – Описание компонентов магазинного автомата

|  |  |  |
| --- | --- | --- |
| Компонента | Определение | Описание |
|  | Начальное состояние автомата | Состояние, которое приобретает автомат в начале своей работы. Представляется в виде стартового правила грамматики (нетерминальный символ А) |
|  | 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 |

Для вывода результата работы синтаксического анализатора нужно использовать флаг m

## 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 PVG-2022 . Данные структуры 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 и формируется дерево разбора. Иначе генерируется исключение.

## 4.6 Структура и перечень сообщений синтаксического анализатора

Индексы ошибок, обнаруживаемых синтаксическим анализатором, находятся в диапазоне 600-609. Перечень сообщений синтаксического анализатора представлен на листинге 4.2.

|  |
| --- |
| ERROR\_ENTRY(600, "[Syntaxis]: Неверная структура программы"),  ERROR\_ENTRY(601, "[Syntaxis]: Ошибочный оператор"),  ERROR\_ENTRY(602, "[Syntaxis]: Ошибка в выражении"),  ERROR\_ENTRY(603, "[Syntaxis]: Ошибка в параметрах функции"),  ERROR\_ENTRY(604, "[Syntaxis]: Ошибка в параметрах вызываемой функции"),  ERROR\_ENTRY(605, "[Syntaxis]: Ошибка знака в выражении"),  ERROR\_ENTRY(606, "[Syntaxis]: Ошибка синтаксического анализа"),  ERROR\_ENTRY(607, "[Syntaxis]: Ошибка уловной конструкции"),  ERROR\_ENTRY(609, "[Syntaxis]: Обнаружена синтаксическая ошибка(смотри журнал Log)"), |

Листинг 4.2 – 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 лексического разбора, а именно дерево разбора и протокол работы автомата с магазинной памятью выводятся в журнал работы программы.

## 4.8. Принцип обработки ошибок

Обработка ошибок происходит следующим образом:

1. Синтаксический анализатор перебирает все правила и цепочки правила грамматики для нахождения подходящего соответствия с конструкцией, представленной в таблице лексем.
2. Если невозможно подобрать подходящую цепочку, то генерируется соответствующая ошибка.
3. В случае ошибки выводится соответствующее сообщение в журнал лога и компилятор прекращает работу.

## 4.9. Контрольный пример

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

# 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 листинге 5.1.

|  |
| --- |
| ERROR\_ENTRY(700, "[Semantic]: Повторное объявление идентификатора"),  ERROR\_ENTRY(701, "[Semantic]: Ошибка в возвращаемом значении"),  ERROR\_ENTRY(702, "[Semantic]: Ошибка в передаваемых значениях в функции: количество параметров не совпадает"),  ERROR\_ENTRY(703, "[Semantic]: Ошибка в передаваемых значениях в функции: типы параметров не совпадают"),  ERROR\_ENTRY(704, "[Semantic]: Нарушены типы данных в выражении"),  ERROR\_ENTRY(705, "[Semantic]: Ошибка экспорта: в библиотеке нет такой функции"),  ERROR\_ENTRY(706, "[Semantic]: Ошибка экспорта: неверные параметры"),  ERROR\_ENTRY(707, "[Semantic]: Ошибка экспорта: ошибочный тип возвращаемого значения"),  ERROR\_ENTRY(708, "[Semantic]: Ошибочный оператор: строки можно только складывать"),  ERROR\_ENTRY(709, "[Semantic]: Ошибочные параметры условной конструкции: строки не могут быть параметрами условной конструкции"),  ERROR\_ENTRY(710, "[Semantic]: Ошибочный оператор: для типа char разрешены только операции + и -"), |

Листинг 5.1 – Перечень сообщений семантического анализатора

## 5.4 Принцип обработки ошибок

При обнаружении хотя бы одной ошибки транслятор завершит свою работу c записью информации об ошибке в лог файл.

## 5.5 Контрольный пример

Результат работы контрольного примера расположен в приложении А, где показан результат лексического анализатора, т.к. представленные таблицы лексем и идентификаторов проходят лексическую и семантическую проверки одновременно.

# 6. Вычисление выражений

## 6.1 Выражения, допускаемые языком

В языке PVG-2022 допускаются выражения, применимые к целочисленным типам данных. В выражениях поддерживаются арифметические операции, такие как +, -, \*, ^, :, /, \ и (), и вызовы функций как операнды арифметических выражений.

Приоритет операций представлен в таблице 6.1.

Таблица 6.1 – Приоритеты операций

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

## 6.2 Польская запись и принцип её построения

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

Польская запись – это альтернативный способ записи арифметических выражений, преимущество которого состоит в отсутствии скобок[3].

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

Алгоритм построения:

– читаем очередной символ;

– если он является идентификатором или литералом, то добавляем его к выходной строке;

– если символ является символом функции, то помещаем его в стек;

– если символ является открывающей скобкой, то она помещается в стек;

– исходная строка просматривается слева направо;

– если символ является закрывающей скобкой, то выталкиваем из стека в выходную строку все символы пока не встретим открывающую скобку. При этом обе скобки удаляются и не попадают в выходную строку;

– как только входная лента закончится все символы из стека выталкиваются в строку;

– в случае если встречаются операции, то выталкиваем из стека в выходную строку все операции, которые имеют выше приоритетность чем последняя операция;

– также, если идентификатор является именем функции, то он заменяется на спецсимвол «@».

Таблица 6.2 – Пример преобразования выражения в обратную польскую запись

|  |  |  |
| --- | --- | --- |
| Исходная строка | Результирующая строка | Стек |
| x+y\*5/(z-2) |  |  |
| +y\*5/(z-2) | x |  |
| y\*5/(z-2) | x | + |
| \*5/(z-2) | xy | + |
| 5/(z-2) | xy | +\* |
| /(z-2) | xy5 | +\* |
| (z-2) | xy5\* | +/ |
| z-2) | xy5\* | +/( |
| -2) | xy5\*z | +/( |
| 2) | xy5\*z | +/(- |
| ) | xy5\*z2 | +/(- |
|  | xy5\*z2- | +/ |
|  | xy5\*z2-/ | + |
|  | xy5\*z2-/+ |  |

Как результат успешного разбора, мы получаем пустой стек и заполненную результирующую строку.

## 6.3 Программная реализация обработки выражений

Пример преобразования выражения к польской записи представлен в таблице 6.2. Преобразование выражений в формат польской записи необходимо для построения более простых алгоритмов их вычисления.

## 6.4 Контрольный пример

Пример преобразования выражения к польской записи представлен в таблице 6.2. Преобразование выражений в формат польской записи необходимо для построения более простых алгоритмов их вычисления.

# 7. Генерация кода

## 7.1 Структура генератора кода

В языке PVG-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 PVG-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 PVG-2022 и на языке ассемблера приведены в таблице 7.1.

Таблица 7.1 – Соответствия типов идентификаторов языка PVG-2022 и языка Ассемблера

|  |  |  |
| --- | --- | --- |
| Тип идентификатора на языке PVG-2022 | Тип идентификатора на языке ассемблера | Пояснение |
| char | BYTE | Хранит символьный тип данных. |
| str | DWORD | Хранит указатель на начало строки. |
| uint | DWORD | Хранит целочисленный тип данных без знака. |
| Лексема | BYTE  DWORD  DWORD | Литералы: символьные,  целочисленные, строковые |

Идентификаторы языка PVG-2022 размещены в сегменте данных (.data). Литералы – в сегменте констант (.const).

## 7.3 Статическая библиотека

В языке PVG-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(C++) | Возвращаемое значение | Описание |
| int ord(char) | char | Возвращает код символа |
| char chr(int) | char | Возвращает символ с заданным кодом |
| int GetMonth(int number) | uint | Возвращает номер месяца |
| int GetDate(int number) | uint | Возвращает дату в формате ДДММГГГГ |
| int GetHours(int number) | uint | Возвращает час |
| int GetMinutes(int number) | uint | Возвращает минуты |

## 7.4 Особенности алгоритма генерации кода

Алгоритм генерации кода выглядит следующим образом:

1. Генерирует заголовочную информацию листинг.7.1: модель памяти, подключение библиотек, прототипы внешних функций, размер стека.

|  |
| --- |
| .586  .model flat, stdcall  includelib libucrt.lib  includelib kernel32.lib  includelib ../Debug/PVG-2022LIB.lib  ExitProcess PROTO :DWORD  GetHours PROTO :DWORD  GetMonth PROTO :DWORD  GetMinutes PROTO :DWORD  GetDate PROTO :DWORD  outputuint PROTO :DWORD  outputchar PROTO :BYTE  outputstring PROTO :DWORD |

Листинг 7.1 –Заголовочная информация

1. Проходит полностью таблицу идентификаторов и заполняет поле .const литералами листинг. 7.2.

|  |
| --- |
| .const  divideOnZeroExeption BYTE "Попытка деления на ноль.", 0 ;STR, для вывода ошибкипри делении на ноль  FindFactor$LEX1 DWORD 1 ;INT  main$LEX4 DWORD 5 ;INT  main$LEX5 BYTE "true", 0 ;STR  main$LEX6 BYTE "Char type:", 0 ;STR  main$LEX7 BYTE 'q' ;CHR  main$LEX8 DWORD 32 ;INT  main$LEX9 BYTE "Number to be circilar shifted:", 0 ;STR  main$LEX10 DWORD 2 ;INT  main$LEX11 BYTE "32<<2:", 0 ;STR  main$LEX13 BYTE "32>>2:", 0 ;STR  main$LEX14 BYTE "Hours:", 0 ;STR  main$LEX16 BYTE "Minutes:", 0 ;STR  main$LEX18 BYTE "Date:", 0 ;STR  main$LEX20 BYTE "Month:", 0 ;STR  main$LEX24 DWORD 0 ;INT |

Листинг 7.2 – Пример заполнения поля .const

## 7.5 Входные параметры генератора кода

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

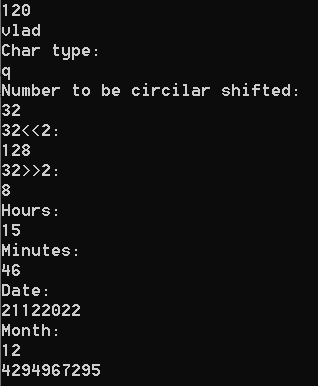


Рисунок 7.2 Результат работы программы на языке PVG-2022

# 8. Тестирование транслятора

## 8.1 Тестирование проверки на допустимость символов

В языке PVG-2022 не разрешается использовать запрещённые входным алфавитом символы где-либо кроме строковых или символьных переменных. Результат использования запрещённого символа показан в таблице 8.1.

Таблица 8.1 – Тестирование фазы проверки на допустимость символов

|  |  |
| --- | --- |
| Исходный код | Диагностическое сообщение |
| main{в} | Ошибка 111: [IN]: Недопустимый символ в исходном файле (-in), строка 1, столбец 4 |

## 8.2 Тестирование лексического анализатора

На этапе лексического анализа могут возникнуть ошибки, описанные в пункте 3.7. Результаты тестирования лексического анализатора показаны в таблице 8.2.

Таблица 8.2 – Тестирование лексического анализатора

|  |  |
| --- | --- |
| Исходный код | Диагностическое сообщение |
| main{  let uint 1s;  } | Ошибка 120: [LA]: Ошибка при разборе токена, строка 2, столбец 11 |
| main{  s = 5;  } | Ошибка 121: [LA]: Используется необъявленный идентификатор, строка 2, столбец 1 |
| let uint func a(){  let q;  } | Ошибка 122: [LA]: Идентификатор не имеет типа, строка 2, столбец 6 |
| let uint func a(){  let uint q;  } | Ошибка 124: [LA]: Отсутствует точка входа |
| main(){  let char a;  }  main(){  let char z;  } | Ошибка 125: [LA]: Обнаружена вторая точка входа, строка 4, столбец 1 |

## 8.3 Тестирование синтаксического анализатора

На этапе синтаксического анализа могут возникнуть ошибки, описанные в пункте 4.6. Результаты тестирования синтаксического анализатора показаны в таблице 8.3.

Таблица 8.3 – Тестирование синтаксического анализатора

|  |  |
| --- | --- |
| Исходный код | Диагностическое сообщение |
| main{  let char a  } | Ошибка 609: [Syntaxis]: Обнаружена синтаксическая ошибка(смотри журнал Log) |

Ошибка синтаксического анализатора также приводит к прекращению выполнения программы и записи соответствующей ошибки в лог журнал.

## 8.4 Тестирование семантического анализатора

Итоги тестирования семантического анализатора приведены в таблице 8.4.

Таблица 8.4 – Тестирование семантического анализатора

|  |  |
| --- | --- |
| Исходный код | Диагностическое сообщение |
| main{  let char a;  let str a;  } | Ошибка 700: [Semantic]: Повторное объявление идентификатора, строка 3, столбец 10 |
| let uint func f(uint q){  let uint a;  return a;  };  main{  let str a;  a = f(3);  return 0;  }; | Ошибка 704: [Semantic]: Нарушены типы данных в выражении, строка 7, столбец 5 |
| let uint func f(uint q){  let str a;  return a;  };  main{  let uint a;  a = f(3);  return 0;  }; | Ошибка 701: [Semantic]: Ошибка в возвращаемом значении, строка 3, столбец 6 |

Продолжение таблицы 8.4 – Тестирование семантического анализатора

|  |  |
| --- | --- |
| Исходный код | Диагностическое сообщение |
| let uint func f(str a){  return 5;  };  main{  let uint a;  a = f(a);  return a;  }; | Ошибка 703: [Semantic]: Ошибка в передаваемых значениях в функции: типы параметров не совпадают, строка 6, столбец 7 |
| let uint func f(uint a, uint b){  return 5;  };  main{  let uint a;  a = f(a);  return a;  }; | Ошибка 702: [Semantic]: Ошибка в передаваемых значениях в функции: количество параметров не совпадает, строка 6, столбец 8 |
| main{  export uint func random(uint max);  let uint a;  a = f(a);  return a;  }; | Ошибка 705: [Semantic]: Ошибка экспорта: в библиотеке нет такой функции, строка 2, столбец 18 |

Ошибка семантического анализатора также приводит к прекращению выполнения программы и записи соответствующей ошибки в лог журнал.

# Заключение

В ходе выполнения курсовой работы был разработан транслятор и генератор кода для языка программирования PVG-2022 со всеми необходимыми компонентами. Таким образом, были выполнены основные задачи данной курсовой работы:

1. Сформулирована спецификация языка PVG-2022;
2. Разработаны конечные автоматы и важные алгоритмы на их основе для эффективной работы лексического анализатора;
3. Осуществлена программная реализация лексического анализатора, распознающего допустимые цепочки спроектированного языка;
4. Разработана контекстно-свободная, приведённая к нормальной форме Грейбах, грамматика для описания синтаксически верных конструкций языка;
5. Осуществлена программная реализация синтаксического анализатора;
6. Разработан семантический анализатор, осуществляющий проверку используемых инструкций на соответствие логическим правилам;
7. Разработан транслятор кода на язык ассемблера;
8. Проведено тестирование всех вышеперечисленных компонентов.

Окончательная версия языка PVG-2022 включает:

По окончании выполнения всех пунктов, изложенных ранее, получили рабочий транслятор языка программирования PVG-2022 на язык ассемблера.

Язык PVG-2022 поддерживает 3 типа данных: беззнаковый целочисленный (uint), символьный(char), логический(bool).

Для целочисленного типа реализована обработка 7 арифметических действий, скобок, обозначающих приоритет операций.

Для символьного типа реализована обработка 2 арифметических действий, скобок, обозначающих приоритет операций.

На этапе семантического анализа производится проверка соответствия исходного кода спецификации по 10 правилам.

Реализованы 3 публичные и 4 приватные функции стандартной библиотеки.

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. Курс лекций по КПО Наркевич А.С.

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

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

|  |
| --- |
| let uint function FindFactorial(uint a){  let uint answer;  answer = 1;  while(a>1)  [  answer = answer\*a;  a = a - 1;  ];  return answer;  };  main  {  export uint function GetHours(uint a);  export uint function GetMonth(uint a);  export uint function GetMinutes(uint a);  export uint function GetDate(uint a);  let uint number;  number = FindFactorial(5);  output number;  let string bl;  bl = "vlad";  output bl;  let char str1;  output "Char type:";  str1 = 'q';  output str1;  let uint demo;  demo = 32;  let uint f;  f = 256;  output "Number to be circilar shifted:";  output demo;  let uint demo1;  demo1 = demo/2;  output "32<<2:";  output demo1;  let uint demo2;  demo2 = demo\2;  output "32>>2:";  output demo2;  let uint number1;  let uint number2;  let uint number3;  let uint number4;  output "Hours:";  number1 = GetHours(1);  output number1;  output "Minutes:";  number2 = GetMinutes(1);  output number2;  output "Date:";  number3 = GetDate(1);  output number3;  output "Month:";  number4 = GetMonth(1);  output number4;  let uint a;  let uint b;  let uint c;  a = 3;  b = 4;  c = a - b;  output c;  return 0;  } |

Листинг 1 – Исходный код программы на языка PVG-2022

|  |
| --- |
| ERROR errors[ERROR\_MAX\_ENTRY] = //таблица ошибок  {  ERROR\_ENTRY(0, "[SYS]: Недопустимый код ошибки"), // код ошибки вне диапазона 0 - ERROR\_MAX\_ENTRY  ERROR\_ENTRY(1, "[SYS]: Системный сбой"), //  ERROR\_ENTRY(2, "[SYS]: Недопустимое количество ошибок"),//  ERROR\_ENTRY(3, "[SYS]: "),  ERROR\_ENTRY\_NODEF(4),  ERROR\_ENTRY\_NODEF(5),  ERROR\_ENTRY\_NODEF(6),  ERROR\_ENTRY\_NODEF(7),  ERROR\_ENTRY\_NODEF(8),  ERROR\_ENTRY\_NODEF(9),  ERROR\_ENTRY\_NODEF10(10),  ERROR\_ENTRY\_NODEF10(20),  ERROR\_ENTRY\_NODEF10(30),  ERROR\_ENTRY\_NODEF10(40),  ERROR\_ENTRY\_NODEF10(50),  ERROR\_ENTRY\_NODEF10(60),  ERROR\_ENTRY\_NODEF10(70),  ERROR\_ENTRY\_NODEF10(80),  ERROR\_ENTRY\_NODEF10(90),  ERROR\_ENTRY(100, "[PARM]: Параметр -in должен быть задан"),//+  ERROR\_ENTRY\_NODEF(101),  ERROR\_ENTRY\_NODEF(102),  ERROR\_ENTRY\_NODEF(103),  ERROR\_ENTRY(104, "[PARM]: Превышена длина входного параметра"),//+  ERROR\_ENTRY\_NODEF(105),  ERROR\_ENTRY\_NODEF(106),  ERROR\_ENTRY\_NODEF(107),  ERROR\_ENTRY\_NODEF(108),  ERROR\_ENTRY\_NODEF(109),  ERROR\_ENTRY(110, "[IN]: Ошибка при открытии файла с исходным кодом (-in)"),//+  ERROR\_ENTRY(111, "[IN]: Недопустимый символ в исходном файле (-in)"),//+  ERROR\_ENTRY(112, "[IN]: Превышен предел размера входного файла (-in)"),//+  ERROR\_ENTRY(113, "[IN]: Превышен предел размера лексемы"),//+  ERROR\_ENTRY(114, "[IN]: Отсуствует закрывающая кавычка"),//+  ERROR\_ENTRY(115, "[IN]: Входная строка превышает максималььный размер"),//+  ERROR\_ENTRY(116, "[IN]: Количество открывающих фигурных скобок не совпадает с количеством закрывающих"),  ERROR\_ENTRY\_NODEF(117),  ERROR\_ENTRY\_NODEF(118),  ERROR\_ENTRY\_NODEF(119),  ERROR\_ENTRY(120, "[LA]: Ошибка при разборе токена"),  ERROR\_ENTRY(121, "[LA]: Используется необъявленный идентификатор"),  ERROR\_ENTRY(122, "[LA]: Итендификатор не имеет типа"),  ERROR\_ENTRY\_NODEF(123),  ERROR\_ENTRY(124, "[LA]: Отсутствует точка входа"),  ERROR\_ENTRY(125, "[LA]: Обнаружена вторая точка входа"),  ERROR\_ENTRY\_NODEF(126),  ERROR\_ENTRY\_NODEF(127),  ERROR\_ENTRY\_NODEF(128),  ERROR\_ENTRY\_NODEF(129),  ERROR\_ENTRY(130, "[ITable]: Ошибка создания таблицы итендификаторов: размер превышает допустимое значение"),//+  ERROR\_ENTRY(131, "[LTable]: Ошибка создания таблицы литералов: размер превышает допустимое значение"),//+  ERROR\_ENTRY(132, "[ITable]: Ошибка добавления в таблицу лексем: превышен макимальный размер таблицы"),//+  ERROR\_ENTRY(133, "[LTable]: Ошибка добавления в таблицу итендификаторов: превышен макимальный размер таблицы"),//+  ERROR\_ENTRY(134, "[ITable]: Ошибка получения элемента из таблицы итендификаторов: элемент не создан или индекс выходит за границы массива"),//+  ERROR\_ENTRY(135, "[LTable]: Ошибка получения элемента из таблицы литералов: элемент не создан или индекс выходит за границы массива"),//+  ERROR\_ENTRY\_NODEF(136),  ERROR\_ENTRY\_NODEF(137),  ERROR\_ENTRY\_NODEF(138),  ERROR\_ENTRY\_NODEF(139),  ERROR\_ENTRY\_NODEF10(140),  ERROR\_ENTRY\_NODEF10(150),  ERROR\_ENTRY\_NODEF10(160),  ERROR\_ENTRY\_NODEF10(170),  ERROR\_ENTRY\_NODEF10(180),  ERROR\_ENTRY\_NODEF10(190),  ERROR\_ENTRY\_NODEF100(200),  ERROR\_ENTRY\_NODEF100(300),  ERROR\_ENTRY\_NODEF100(400),  ERROR\_ENTRY\_NODEF100(500),  ERROR\_ENTRY(600, "[Syntaxis]: Неверная структура программы"),  ERROR\_ENTRY(601, "[Syntaxis]: Ошибочный оператор"),  ERROR\_ENTRY(602, "[Syntaxis]: Ошибка в выражении"),  ERROR\_ENTRY(603, "[Syntaxis]: Ошибка в параметрах функции"),  ERROR\_ENTRY(604, "[Syntaxis]: Ошибка в параметрах вызываемой функции"),  ERROR\_ENTRY(605, "[Syntaxis]: Ошибка знака в выражении"),  ERROR\_ENTRY(606, "[Syntaxis]: Ошибка синтаксического анализа"),  ERROR\_ENTRY(607, "[Syntaxis]: Ошибка уловной конструкции"),  ERROR\_ENTRY\_NODEF(608),  ERROR\_ENTRY(609, "[Syntaxis]: Обнаружена синтаксическая ошибка(смотри журнал Log)"),  ERROR\_ENTRY\_NODEF10(610),  ERROR\_ENTRY\_NODEF10(620),  ERROR\_ENTRY\_NODEF10(630),  ERROR\_ENTRY\_NODEF10(640),  ERROR\_ENTRY\_NODEF10(650),  ERROR\_ENTRY\_NODEF10(660),  ERROR\_ENTRY\_NODEF10(670),  ERROR\_ENTRY\_NODEF10(680),  ERROR\_ENTRY\_NODEF10(690),  ERROR\_ENTRY(700, "[Semantic]: Повторное объявление идентификатора"),  ERROR\_ENTRY(701, "[Semantic]: Ошибка в возвращаемом значении"),  ERROR\_ENTRY(702, "[Semantic]: Ошибка в передаваемых значениях в функции: количество параметров не совпадает"),  ERROR\_ENTRY(703, "[Semantic]: Ошибка в передаваемых значениях в функции: типы параметров не совпадают"),  ERROR\_ENTRY(704, "[Semantic]: Нарушены типы данных в выражении"),  ERROR\_ENTRY(705, "[Semantic]: Ошибка экспорта: в библиотеке нет такой функции"),  ERROR\_ENTRY(706, "[Semantic]: Ошибка экспорта: неверные параметры"),  ERROR\_ENTRY(707, "[Semantic]: Ошибка экспорта: ошибочный тип возвращаемого значения"),  ERROR\_ENTRY(708, "[Semantic]: Ошибочный оператор: строки можно только складывать"),  ERROR\_ENTRY(709, "[Semantic]: Ошибочные параметры условной конструкции: строки не могут быть параметрами условной конструкции"),  ERROR\_ENTRY(710, "[Semantic]: Ошибочный оператор: для типа char разрешены только операции + и -"),  ERROR\_ENTRY\_NODEF10(720),ERROR\_ENTRY\_NODEF10(730),ERROR\_ENTRY\_NODEF10(740),  ERROR\_ENTRY\_NODEF10(750),ERROR\_ENTRY\_NODEF10(760),ERROR\_ENTRY\_NODEF10(770),ERROR\_ENTRY\_NODEF10(780),  ERROR\_ENTRY\_NODEF10(790),  ERROR\_ENTRY\_NODEF100(800), ERROR\_ENTRY\_NODEF100(900)  }; |

Листинг 2 – Таблица ошибок языка PVG-2022

# Приложение Б

Листинг 1 Таблица идентификаторов контрольного примера

|  |
| --- |
| --------Таблица идентификаторов--------  Позиция Имя Вид Тип данных Значение  0 FindFactor Функция Число 0  1 FindFactora Параметр Число 0  2 FindFactoranswerПеременная Число 0  3 FindFactor$LEX1 Литерал Число 1  4 GetHours Функция Число 0  5 mainGetHoursa Параметр Число 0  6 GetMonth Функция Число 0  7 mainGetMontha Параметр Число 0  8 GetMinutes Функция Число 0  9 mainGetMinutesa Параметр Число 0  10 GetDate Функция Число 0  11 mainGetDatea Параметр Число 0  12 mainnumber Переменная Число 0  13 main$LEX4 Литерал Число 5  14 mainbl Переменная Строка \*пустая строка\*  15 main$LEX5 Литерал Строка "vlad"  16 mainstr1 Переменная Символ  17 main$LEX6 Литерал Строка "Char type:"  18 main$LEX7 Литерал Символ q  19 maindemo Переменная Число 0  20 main$LEX8 Литерал Число 32  21 mainf Переменная Число 0  22 main$LEX9 Литерал Число 256  23 main$LEX10 Литерал Строка "Number to be circilar shifted:"  24 maindemo1 Переменная Число 0  25 main$LEX11 Литерал Число 2  26 main$LEX12 Литерал Строка "32<<2:"  27 maindemo2 Переменная Число 0  28 main$LEX14 Литерал Строка "32>>2:"  29 mainnumber1 Переменная Число 0  30 mainnumber2 Переменная Число 0  31 mainnumber3 Переменная Число 0  32 mainnumber4 Переменная Число 0  33 main$LEX15 Литерал Строка "Hours:"  34 main$LEX17 Литерал Строка "Minutes:"  35 main$LEX19 Литерал Строка "Date:"  36 main$LEX21 Литерал Строка "Month:"  37 maina Переменная Число 0  38 mainb Переменная Число 0  39 mainc Переменная Число 0  40 main$LEX23 Литерал Число 3  41 main$LEX24 Литерал Число 4 |

Листинг 1 – Таблица идентификаторов контрольного примера

# Приложение В

|  |
| --- |
| Greibach greibach(NS('S'), TS('$'),  9,  Rule(NS('S'), GRB\_ERROR\_SERIES + 0,  6, //todo m{NrE;}; tfi(F){NrE;};S m{NrE;};S tfi(F){NrE;};  Rule::Chain(7, TS('m'), TS('{'), NS('N'), TS('r'), NS('E'), TS(';'), TS('}')),//+  Rule::Chain(6, TS('m'), TS('{'), TS('r'), NS('E'), TS(';'), TS('}')),//+  Rule::Chain(15, TS('d'), TS('t'), TS('f'), TS('i'), TS('('), NS('F'), TS(')'), TS('{'), NS('N'), TS('r'), NS('E'), TS(';'), TS('}'), TS(';'), NS('S')),//+  Rule::Chain(14, TS('d'), TS('t'), TS('f'), TS('i'), TS('('), NS('F'), TS(')'), TS('{'), TS('r'), NS('E'), TS(';'), TS('}'), TS(';'), NS('S')),  Rule::Chain(14, TS('d'), TS('t'), TS('f'), TS('i'), TS('('), TS(')'), TS('{'), NS('N'), TS('r'), NS('E'), TS(';'), TS('}'), TS(';'), NS('S')),  Rule::Chain(13, TS('d'), TS('t'), TS('f'), TS('i'), TS('('), TS(')'), TS('{'), TS('r'), NS('E'), TS(';'), TS('}'), TS(';'), NS('S'))  ),  Rule(NS('N'), GRB\_ERROR\_SERIES + 1,  14, //todo dti; rE; i=E; dtfi(F); dti;N rE;N i=E;N dtfi(F);N pl;N pi;N pl; pi; pi(W);  Rule::Chain(4, TS('d'), TS('t'), TS('i'), TS(';')),//+  Rule::Chain(5, TS('d'), TS('t'), TS('i'), TS(';'), NS('N')), //+  Rule::Chain(4, TS('i'), TS('v'), NS('E'), TS(';')), //+  Rule::Chain(5, TS('i'), TS('v'), NS('E'), TS(';'), NS('N')),  Rule::Chain(9, TS('e'), TS('t'), TS('f'), TS('i'), TS('('), NS('F'), TS(')'), TS(';'), NS('N')), //+  Rule::Chain(8, TS('e'), TS('t'), TS('f'), TS('i'), TS('('), NS('F'), TS(')'), TS(';')),//+  Rule::Chain(8, TS('e'), TS('t'), TS('f'), TS('i'), TS('('), TS(')'), TS(';'), NS('N')),  Rule::Chain(7, TS('e'), TS('t'), TS('f'), TS('i'), TS('('), TS(')'), TS(';')),  Rule::Chain(4, TS('p'), TS('i'), TS(';'), NS('N')),//+  Rule::Chain(3, TS('p'), TS('i'), TS(';')),//+  Rule::Chain(4, TS('p'), TS('l'), TS(';'), NS('N')),//+  Rule::Chain(3, TS('p'), TS('l'), TS(';')),//+  Rule::Chain(8, TS('u'), TS('('), NS('B'), TS(')'), TS('['), NS('N'), TS(']'), TS(';')),  Rule::Chain(9, TS('u'), TS('('), NS('B'), TS(')'), TS('['), NS('N'), TS(']'), TS(';'), NS('N'))  ),  Rule(NS('E'), GRB\_ERROR\_SERIES + 2,  9, //todo i l (E) i(W) iM lM (E)M i(W)M  Rule::Chain(1, TS('i')),  Rule::Chain(1, TS('l')),  Rule::Chain(3, TS('('), NS('E'), TS(')')),//+  Rule::Chain(4, TS('i'), TS('('), NS('W'), TS(')')),  Rule::Chain(3, TS('i'), TS('('), TS(')')),  Rule::Chain(2, TS('i'), NS('M')),//+  Rule::Chain(2, TS('l'), NS('M')),//+  Rule::Chain(4, TS('('), NS('E'), TS(')'), NS('M')),//+  Rule::Chain(5, TS('i'), TS('('), NS('W'), TS(')'), NS('M'))  ),  Rule(NS('M'), GRB\_ERROR\_SERIES + 3,  4, //todo vE vEM v(E) v(E)M  Rule::Chain(2, TS('v'), NS('E')),//+  Rule::Chain(4, TS('v'), TS('('), NS('E'), TS(')')),  Rule::Chain(5, TS('v'), TS('('), NS('E'), TS(')'), NS('M')),  Rule::Chain(3, TS('v'), NS('E'), NS('M'))  ),  Rule(NS('F'), GRB\_ERROR\_SERIES + 4,  2, //todo ti ti,F  Rule::Chain(2, TS('t'), TS('i')),//+  Rule::Chain(4, TS('t'), TS('i'), TS(','), NS('F'))//+  ),  Rule(NS('W'), GRB\_ERROR\_SERIES + 5,  4, //todo i l i,W l,W  Rule::Chain(1, TS('i')),  Rule::Chain(1, TS('l')),  Rule::Chain(3, TS('i'), TS(','), NS('W')),  Rule::Chain(3, TS('l'), TS(','), NS('W'))  ),  Rule(NS('B'), GRB\_ERROR\_SERIES + 6,  3, //todo ibi ibl lbi lbl  Rule::Chain(3, TS('i'), TS('b'), TS('i')),  Rule::Chain(3, TS('i'), TS('b'), TS('l')),  Rule::Chain(3, TS('l'), TS('b'), TS('i')),  Rule::Chain(3, TS('l'), TS('b'), TS('l'))  )  ); |

Листинг 1 – Грамматика языка -2022

|  |
| --- |
| struct Mfst //магазинный автомат  {  enum RC\_STEP {  NS\_OK,  NS\_NORULE,  NS\_NORULECHAIN,  NS\_ERROR,  TS\_OK,  TS\_NOK,  LENTA\_END,  SURPRISE  };  struct MfstDiagnosis  {  short lenta\_position;  RC\_STEP rc\_step;  short nrule;  short nrule\_chain;  MfstDiagnosis();  MfstDiagnosis(  short plenta\_position,  RC\_STEP prc\_step,  short pnrule,  short pnrule\_chain  );  } diagnosis[MFST\_DIAGN\_NUMBER];  GRBALPHABET\* lenta;  short lenta\_position;  short nrule;  short nrulechain;  short lenta\_size;  GRB::Greibach greibach;  LEX::LEX lex;  MFSTSTACK st;  bool shallWrite;  use\_container<std::stack<MfstState>> storestate;  Mfst();  Mfst(  LEX::LEX plex,  GRB::Greibach pgreibach,  bool shouldWrite  );  char\* getCSt(char\* buf);  char\* getCLenta(char\* buf, short pos, short n = 25);  char\* getDiagnosis(short n, char\* buf);  bool savestate(const Log::LOG& log);  bool reststate(const Log::LOG& log);  bool push\_chain(  GRB::Rule::Chain chain  );  RC\_STEP step(const Log::LOG& log);  bool start(const Log::LOG& log);  bool savediagnosis(  RC\_STEP pprc\_step  );  void printrules(const Log::LOG& lo);  struct Deducation  {  short size;  short\* nrules;  short\* nrulechains;  Deducation() { size = 0; nrules = 0; nrulechains = 0; };  } deducation;  bool savededucation(); |

Листинг 2 – Структура магазинного автомата

struct Greibach

{

short size;

GRBALPHABET startN;

GRBALPHABET stbottomT;

Rule\* rules;

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

Greibach(GRBALPHABET pstartN, GRBALPHABET pstbootomT, short psize, Rule r,);

short getRule(GRBALPHABET pnn, Rule& prule);

Rule getRule(short n);

};

Greibach getGreibach();

Листинг 3 – Структура грамматики Грейбах

|  |
| --- |
| Шаг :Правило Входная лента Стек  0 :S->dtfi(F){NrE;};S dtfi(ti){dti;ivl;u(ibl)[i S$  1 : SAVESTATE: 1  1 : dtfi(ti){dti;ivl;u(ibl)[i dtfi(F){NrE;};S$  2 : tfi(ti){dti;ivl;u(ibl)[iv tfi(F){NrE;};S$  3 : fi(ti){dti;ivl;u(ibl)[ivi fi(F){NrE;};S$  4 : i(ti){dti;ivl;u(ibl)[iviv i(F){NrE;};S$  5 : (ti){dti;ivl;u(ibl)[ivivi (F){NrE;};S$  6 : ti){dti;ivl;u(ibl)[ivivi; F){NrE;};S$  7 :F->ti ti){dti;ivl;u(ibl)[ivivi; F){NrE;};S$  8 : SAVESTATE: 2  8 : ti){dti;ivl;u(ibl)[ivivi; ti){NrE;};S$  9 : i){dti;ivl;u(ibl)[ivivi;i i){NrE;};S$  10 : ){dti;ivl;u(ibl)[ivivi;iv ){NrE;};S$  11 : {dti;ivl;u(ibl)[ivivi;ivi {NrE;};S$  12 : dti;ivl;u(ibl)[ivivi;iviv NrE;};S$  13 :N->dti; dti;ivl;u(ibl)[ivivi;iviv NrE;};S$  14 : SAVESTATE: 3  14 : dti;ivl;u(ibl)[ivivi;iviv dti;rE;};S$  15 : ti;ivl;u(ibl)[ivivi;ivivl ti;rE;};S$  16 : i;ivl;u(ibl)[ivivi;ivivl; i;rE;};S$  17 : ;ivl;u(ibl)[ivivi;ivivl;] ;rE;};S$  18 : ivl;u(ibl)[ivivi;ivivl;]; rE;};S$  19 : TS\_NOK/NS\_NORULECHAIN  19 : RESSTATE  19 : dti;ivl;u(ibl)[ivivi;iviv NrE;};S$  20 :N->dti;N dti;ivl;u(ibl)[ivivi;iviv NrE;};S$  21 : SAVESTATE: 3  21 : dti;ivl;u(ibl)[ivivi;iviv dti;NrE;};S$  22 : ti;ivl;u(ibl)[ivivi;ivivl ti;NrE;};S$  23 : i;ivl;u(ibl)[ivivi;ivivl; i;NrE;};S$  24 : ;ivl;u(ibl)[ivivi;ivivl;] ;NrE;};S$  25 : ivl;u(ibl)[ivivi;ivivl;]; NrE;};S$ |

Листинг 4 – Разбор исходного кода синтаксическим анализатором

# Приложение Г

|  |
| --- |
| bool NotaciaPolska(int lextable\_pos, LT::LexTable& lextable, IT::IdTable& idtable)  {  container<std::stack<char>> stack;  std::string PolishString;  std::vector<int> ids;  int operators\_count = 0, operands\_count = 0, iterator = 0, right\_counter = 0, left\_counter = 0, params\_counter = 0;  for (int i = lextable\_pos; i < lextable.size; i++, iterator++) {  char lexem = lextable.table[i].lexema;  char data = lextable.table[i].data;  size\_t stack\_size = stack.size();  if (idtable.table[lextable.table[i].idxTI].idtype == IT::IDTYPE::F) {  stack.push('@');  operands\_count--;  }  switch (lexem) {  case LEX\_OPERATOR:  {  if (!stack.empty() && stack.top() != LEX\_LEFTHESIS) {  while (!stack.empty() && get\_priority(data) <= get\_priority(stack.top())) {  PolishString += stack.top();  stack.pop();  }  }  stack.push(data);  operators\_count++;  break;  }  case LEX\_COMMA:  {  while (!stack.empty()) {  if (stack.top() == LEX\_LEFTHESIS)  break;  PolishString += stack.top();  stack.pop();  }  operands\_count--;  break;  }  case LEX\_LEFTHESIS:  {  left\_counter++;  stack.push(lexem);  break;  }  case LEX\_RIGHTHESIS:  {  right\_counter++;  if (!find\_elem(stack, stack\_size, LEX\_LEFTHESIS))  return false;  while (stack.top() != LEX\_LEFTHESIS) {  PolishString += stack.top();  stack.pop();  }  stack.pop();  if (!stack.empty() && stack.top() == '@') {  PolishString += stack.top() + toString(params\_counter - 1);  params\_counter = 0;  stack.pop();  }  break;  }  case LEX\_SEMICOLON:  {  if (operators\_count != 0 && operands\_count != 0)  if ((!stack.empty() && (stack.top() == LEX\_RIGHTHESIS || stack.top() == LEX\_LEFTHESIS))  || right\_counter != left\_counter || operands\_count - operators\_count != 1)  return false;  while (!stack.empty()) {  PolishString += stack.top();  stack.pop();  }  fixIt(lextable, PolishString, iterator, lextable\_pos, ids);  return true;  break;  }  case LEX\_ID: {  if (std::find(stack.c.begin(), stack.c.begin(), '@') != stack.c.end())  params\_counter++;  PolishString += lexem;  if (lextable.table[i].idxTI != LT\_TI\_NULLIDX)  ids.push\_back(lextable.table[i].idxTI);  operands\_count++;  break;  }  case LEX\_LITERAL: {  if (std::find(stack.c.begin(), stack.c.begin(), '@') != stack.c.end())  params\_counter++;  PolishString += lexem;  if (lextable.table[i].idxTI != LT\_TI\_NULLIDX)  ids.push\_back(lextable.table[i].idxTI);  operands\_count++;  break;  }  }  }  return true;  } |

Листинг 1 – Программная реализация механизма преобразования в ПОЛИЗ

# Приложение Д

|  |
| --- |
| .586  .model flat, stdcall  includelib libucrt.lib  includelib kernel32.lib  includelib ../Debug/PVG-2022LIB.lib  ExitProcess PROTO :DWORD  GetHours PROTO :DWORD  GetMonth PROTO :DWORD  GetMinutes PROTO :DWORD  GetDate PROTO :DWORD  outputuint PROTO :DWORD  outputchar PROTO :BYTE  outputstr PROTO :DWORD  .stack 4096  .const  divideOnZeroExeption BYTE "Попытка деления на ноль.", 0 ;STR, для вывода ошибки при делении на ноль  FindFactor$LEX1 DWORD 1 ;INT  main$LEX4 DWORD 5 ;INT  main$LEX5 BYTE "vlad", 0 ;STR  main$LEX6 BYTE "Char type:", 0 ;STR  main$LEX7 BYTE 'q' ;CHR  main$LEX8 DWORD 32 ;INT  main$LEX9 DWORD 254 ;INT  main$LEX10 BYTE "Number to be circilar shifted:", 0 ;STR  main$LEX11 DWORD 2 ;INT  main$LEX12 BYTE "32<<2:", 0 ;STR  main$LEX14 BYTE "32>>2:", 0 ;STR  main$LEX15 BYTE "Hours:", 0 ;STR  main$LEX17 BYTE "Minutes:", 0 ;STR  main$LEX19 BYTE "Date:", 0 ;STR  main$LEX21 BYTE "Month:", 0 ;STR  main$LEX23 DWORD 3 ;INT  main$LEX24 DWORD 4 ;INT  main$LEX25 DWORD 0 ;INT  .data  FindFactoranswer DWORD 0 ;INT  mainnumber DWORD 0 ;INT  mainbl DWORD 0 ;STR  mainstr1 BYTE 0 ;CHR  maindemo DWORD 0 ;INT  mainf DWORD 0 ;INT  maindemo1 DWORD 0 ;INT  maindemo2 DWORD 0 ;INT  mainnumber1 DWORD 0 ;INT  mainnumber2 DWORD 0 ;INT  mainnumber3 DWORD 0 ;INT  mainnumber4 DWORD 0 ;INT  maina DWORD 0 ;INT  mainb DWORD 0 ;INT  mainc DWORD 0 ;INT  .code  $FindFactor PROC uses ebx ecx edi esi , FindFactora: DWORD  ; String #3 :ivl  push FindFactor$LEX1  pop FindFactoranswer  While17Start:  mov eax, FindFactora  mov ebx, FindFactor$LEX1  cmp eax, ebx  jl While17End  ; String #6 :iviiv  push FindFactoranswer  push FindFactora  pop ebx  pop eax  mul ebx  push eax  pop FindFactoranswer  ; String #7 :ivilv  push FindFactora  push FindFactor$LEX1  pop ebx  pop eax  sub eax, ebx  push eax  pop FindFactora  jmp While17Start  While17End:  mov eax, FindFactoranswer  ret  $FindFactor ENDP  main PROC  ; String #18 :ivil@1  invoke $FindFactor, main$LEX4  push eax ;результат функции  pop mainnumber  push mainnumber  CALL outputuint  ; String #21 :ivl  push offset main$LEX5  pop mainbl  push mainbl  CALL outputstr  push offset main$LEX6  CALL outputstr  ; String #25 :ivl  movzx eax, main$LEX7  push eax  pop eax  mov mainstr1, al  push eax  movzx eax, mainstr1  push eax  CALL outputchar  pop eax  ; String #28 :ivl  push main$LEX8  pop maindemo  ; String #30 :ivl  push main$LEX9  pop mainf  push offset main$LEX10  CALL outputstr  push maindemo  CALL outputuint  ; String #34 :ivilv  push maindemo  push main$LEX11  pop ebx  pop eax  push ecx ; сохраняем данные регистра ecx  mov ecx, ebx  SHL eax, cl  pop ecx  push eax  pop maindemo1  push offset main$LEX12  CALL outputstr  push maindemo1  CALL outputuint  ; String #38 :ivilv  push maindemo  push main$LEX11  pop ebx  pop eax  push ecx ; сохраняем данные регистра ecx  mov ecx, ebx  SHR eax, cl  pop ecx  push eax  pop maindemo2  push offset main$LEX14  CALL outputstr  push maindemo2  CALL outputuint  push offset main$LEX15  CALL outputstr  ; String #46 :ivil@1  invoke GetHours, FindFactor$LEX1  push eax ;результат функции  pop mainnumber1  push mainnumber1  CALL outputuint  push offset main$LEX17  CALL outputstr  ; String #49 :ivil@1  invoke GetMinutes, FindFactor$LEX1  push eax ;результат функции  pop mainnumber2  push mainnumber2  CALL outputuint  push offset main$LEX19  CALL outputstr  ; String #52 :ivil@1  invoke GetDate, FindFactor$LEX1  push eax ;результат функции  pop mainnumber3  push mainnumber3  CALL outputuint  push offset main$LEX21  CALL outputstr  ; String #55 :ivil@1  invoke GetMonth, FindFactor$LEX1  push eax ;результат функции  pop mainnumber4  push mainnumber4  CALL outputuint  ; String #60 :ivl  push main$LEX23  pop maina  ; String #61 :ivl  push main$LEX24  pop mainb  ; String #62 :iviiv  push maina  push mainb  pop ebx  pop eax  sub eax, ebx  push eax  pop mainc  push mainc  CALL outputuint  mov eax, main$LEX25  jmp endPoint  div\_by\_0:  push offset divideOnZeroExeption  CALL outputstr  endPoint:  invoke ExitProcess, eax  main ENDP  end main |

Листинг 1 – Результат генерации кода контрольного примера в Ассемблере