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 SKG-2023»

Выполнил студент Сподобаев Кирилл Геннадьевич

(Ф.И.О.)

Руководитель проекта асс. Мущук Артур Николаевич

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

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

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

Консультант асс. Мущук Артур Николаевич

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

Курсовой проект защищен с оценкой

Минск 2023

**Содержание**

[**Введение** 4](#_Toc153889619)

[**Глава 1. Спецификация языка программирования** 5](#_Toc153889620)

[**1.1.** **Характеристика языка программирования** 5](#_Toc153889621)

[**1.2.** **Алфавит языка** 5](#_Toc153889622)

[**1.3.** **Символы сепараторы** 5](#_Toc153889623)

[**1.4.** **Применяемые кодировки** 6](#_Toc153889624)

[**1.5.** **Типы данных** 6](#_Toc153889625)

[**1.6.** **Преобразование типов данных** 7](#_Toc153889626)

[**1.7.** **Идентификаторы** 7](#_Toc153889627)

[**1.8.** **Литералы** 7](#_Toc153889628)

[**1.9.** **Область видимости** 7](#_Toc153889629)

[**1.10.** **Инициализация данных** 8](#_Toc153889630)

[**1.11.** **Инструкции языка** 8](#_Toc153889631)

[**1.12.** **Операции языка** 9](#_Toc153889632)

[**1.13.** **Выражения и их вычисления** 9](#_Toc153889634)

[**1.14.** **Программные конструкции языка** 9](#_Toc153889635)

[**1.15.** **Область видимости идентификаторов** 10](#_Toc153889636)

[**1.16.** **Семантические проверки** 10](#_Toc153889637)

[**1.17.** **Распределение оперативной памяти на этапе выполнения** 10](#_Toc153889638)

[**1.18.** **Стандартная библиотека и её состав** 10](#_Toc153889639)

[**1.19.** **Ввод и вывод данных** 11](#_Toc153889640)

[**1.20.** **Точка входа** 11](#_Toc153889641)

[**1.21.** **Препроцессор** 11](#_Toc153889642)

[**1.22.** **Соглашения о вызовах** 11](#_Toc153889643)

[**1.23.** **Объектный код** 11](#_Toc153889644)

[**1.24.** **Классификация сообщений транслятора** 11](#_Toc153889645)

[**1.25.** **Контрольный пример** 11](#_Toc153889646)

[**Глава 2. Структура транслятора** 12](#_Toc153889647)

[**2.1 Компоненты транслятора, их назначение и принципы взаимодействия** 12](#_Toc153889648)

[**2.2 Перечень входных параметров транслятора** 13](#_Toc153889649)

[**2.3 Перечень протоколов, формируемых транслятором и их содержимое** 13](#_Toc153889650)

[**Глава 3. Разработка лексического анализатора** 14](#_Toc153889651)

[**3.1 Структура лексического анализатора** 14](#_Toc153889652)

[**3.3 Параметры лексического анализатора и режимы его работы** 14](#_Toc153889653)

[**3.4 Алгоритм лексического анализа** 14](#_Toc153889654)

[**3.5 Контроль входных символов** 15](#_Toc153889655)

[**3.6 Удаление избыточных символов** 16](#_Toc153889656)

[**3.7 Перечень ключевых слов, сепараторов, символов операций и соответствующих им лексем, регулярных выражений и конечных автоматов** 16](#_Toc153889657)

[**3.8 Основные структуры данных** 17](#_Toc153889658)

[**3.9 Структура и перечень сообщений лексического анализатора** 19](#_Toc153889659)

[**3.10 Принцип обработки ошибок** 19](#_Toc153889660)

[**3.11 Контрольный пример** 20](#_Toc153889661)

[**Глава 4. Разработка синтаксического анализатора** 21](#_Toc153889662)

[**4.1 Структура синтаксического анализатора** 21](#_Toc153889663)

[**4.2 Контекстно свободная грамматика, описывающая синтаксис языка** 21](#_Toc153889664)

[**4.3 Построение конечного магазинного автомата** 23](#_Toc153889665)

[**4.4 Основные структуры данных** 24](#_Toc153889666)

[**4.5 Описание алгоритма синтаксического разбора** 24](#_Toc153889667)

[**4.6 Параметры синтаксического анализатора и режимы его работы** 25](#_Toc153889668)

[**4.7 Структура и перечень сообщений синтаксического анализатора** 25](#_Toc153889669)

[**4.8 Принцип обработки ошибок** 25](#_Toc153889670)

[**4.9 Контрольный пример** 25](#_Toc153889671)

[**Глава 5. Разработка семантического анализатора** 26](#_Toc153889672)

[**5.1 Структура семантического анализатора** 26](#_Toc153889673)

[**5.2 Функции семантического анализатора** 26](#_Toc153889674)

[**5.3 Структура и перечень сообщений семантического анализатора** 26](#_Toc153889675)

[**5.4 Принцип обработки ошибок** 26](#_Toc153889676)

[**5.5 Контрольный пример** 27](#_Toc153889677)

[**Глава 6. Преобразование выражений** 28](#_Toc153889678)

[**6.1 Выражения, допускаемые языком** 28](#_Toc153889679)

[**6.2 Польская запись** 28](#_Toc153889680)

[**6.3 Программная реализация обработки выражений** 29](#_Toc153889681)

[**6.4 Контрольный пример** 29](#_Toc153889682)

[**Глава 7. Генерация кода** 30](#_Toc153889683)

[**7.1 Структура генератора кода** 30](#_Toc153889684)

[**7.2 Представление типов данных в оперативной памяти** 31](#_Toc153889685)

[**7.3 Статическая библиотека** 31](#_Toc153889686)

[**Глава 8. Тестирование транслятора** 33](#_Toc153889687)

[**8.1Тестирование лексического анализатора** 33](#_Toc153889688)

[**8.2 Тестирование синтаксического анализатора** 33](#_Toc153889689)

[**8.3 Тестирование семантического анализатора** 33](#_Toc153889690)

[**Приложение А** 35](#_Toc153889691)

[**Приложение Б** 36](#_Toc153889692)

[**Приложение В** 40](#_Toc153889693)

[**Приложение Г** 41](#_Toc153889695)

[**Приложение Д** 44](#_Toc153889696)

[**Приложение Е** 47](#_Toc153889697)

[**Приложение Ж** 51](#_Toc153889698)

[**Литература** 54](#_Toc153889699)

# **Введение**

Задачей данного курсового проекта является разработка транслятора для своего языка программирования: SKG-2023. Этот язык программирования предназначен для выполнения простейших операций и арифметических действий над строками и числами.

Главная задача транслятора заключается в том, чтобы сделать программу, написанную языке программирования SKG-2023, понятной компьютеру. В данном курсовом проекте мой исходный код транслируется на язык ассемблера. Язык ассемблера – тип языка программирования низкого уровня, представляющий собой формат записи машинных команд, удобный для восприятия человеком.

Исходя из цели курсового проекта, были определены следующие задачи:

– разработка спецификации языка программирования;

– разработка структуры транслятора;

– разработка лексического анализатора;

– разработка синтаксического анализатора;

– разработка семантического анализатора;

– обработка выражений;

– генерация кода на язык Assembler;

– тестирование транслятора.

Решения каждой из поставленных задач буду приведены в соответствующих главах курсового проекта:

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.

# **Глава 1. Спецификация языка программирования**

## **Характеристика языка программирования**

Язык программирования SKG-2023 – это процедурный язык высокого уровня, который транслируется в язык ассемблера. Строго типизируемый.

* 1. **Алфавит языка**

В основе алфавита SKG-2023 лежит таблица символов WINDOWS-1251. Исходный код SKG-2023 может содержать символы латинского алфавита верхнего и малого регистров, цифры десятичной системы счисления от 0 до 9.

Таблица 1.1 – Алфавит языка

|  |  |
| --- | --- |
| Название подгруппы | Символы подгруппы |
| Символы латинского языка и кириллицы | [a-z|A-Z|А-Я|а-я]\* |
| Знаковые символы и числовые символы | [0-9|?|!|;|:|-|+|\*|/|%]\* |

## **Символы сепараторы**

Сепараторы необходимы для разделения операций языка. Сепараторы, используемые в языке программирования SKG-2023, приведены в таблице 1.2.

Таблица 1.2 – 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 |
| ‘…’ | Одинарные кавычки | Символьный литерал |
| = | Знак «равно» | Присваивание значения |
| , | Запятая | Разделение параметров |
| : | Двоеточие |  |

## **Применяемые кодировки**

В основе алфавита SKG-2023 лежит таблица символов WINDOWS-1251. Исходный код SKG-2023 может содержать символы кириллицы, латинского алфавита верхнего и малого регистров, цифры десятичной системы счисления от 0 до 9, спецсимволы.

## **Типы данных**

В языке используется 3 основных типов данных, которые описываются в таблице 1.3.

Таблица 1.3 – Типы данных языка SKG-2023

|  |  |
| --- | --- |
| Тип данных | Описание типа данных |
| int | Фундаментальный тип данных. Предусмотрен для объявления целочисленных знаковых данных (4 байта).  Автоматически инициализируется нулевым значением.  Возможные операции:  *Арифметические:*  + – бинарный, суммирование;  - – бинарный, вычитание;  \* – бинарный, умножение;  / – бинарный, деление;  % – Бинарный, oстаток от деления; |
| symbol | Фундаментальный тип данных. Предусмотрен для объявления символов (2 байтa). Автоматическая инициализация нулевым символом. |
| boolean | Фундаментальный тип данных (2 байта), используемый для объявления логической переменной, которая принимает одно из двух значений: true или false. Без явно указанной инициализации переменной, присваивается нулевое значение (false). |

## **Преобразование типов данных**

В языке программирования SKG-2023 преобразование типов данных не поддерживается.

## **Идентификаторы**

Для именования функций, параметров и переменных используются идентификаторы. Идентификаторы, объявленные внутри функционального блока, получают префикс, идентичный имени функции, внутри которой они объявлены. Идентификаторы не должны совпадать с ключевыми словами. Не предусмотрены зарезервированные идентификаторы. Имя идентификатора составляется по следующим образом:

* состоит из символов латинского алфавита [A-Z|a-z]+.
* максимальная длина идентификатора равна 15 и не должна превышать это значение.

## **Литералы**

В языке программирования SKG-2023 существует только 3 типа литералов: целые, символьные и логические. Их краткое описание представлено в таблице 1.4.

Таблица 1.4 – Литералы

|  |  |
| --- | --- |
| Тип литерала | Описание |
| Литералы целого типа | Целочисленные литералы. Могут быть представлены как в десятичном, двоичном (первый символ: ^), так и в восьмеричном (первый символ: 0)представлении. |
| Символьные литералы | Символы, заключённые в ' ' (одинарные кавычки), инициализируются нулевым символом, строковые переменные. |
| Логические литералы | true и false являются логическими литералами. true интерпретируется, как 1, а false как 0. Логический тип относится к целым типам. |

## **Область видимости**

В языке программирования SKG-2023 переменные обязаны находиться внутри программного блока функций. Внутри разных областей видимости разрешено объявление переменных с одинаковыми именами. Все переменные, параметры или функции внутри области видимости получают префикс, который отображается в таблице идентификаторов.

## **Инициализация данных**

Способы инициализации переменных языка программирования SKG-2023 представлены в таблице 1.5.

Таблица 1.5 – Способы инициализации переменных

|  |  |
| --- | --- |
| Вид инициализации | Примечание |
| var <тип данных> <идентификатор>; | Объявление(по умолчанию) |
| var <тип данных> <идентификатор> = <значение>; | Инициализация переменной с присваиванием значения. |

## **Инструкции языка**

Инструкции языка программирования SKG-2023 представлены в общем виде в таблице 1.6.

Таблица 1.6 – Инструкции языка программирования SKG-2023

|  |  |
| --- | --- |
| Инструкция | Запись |
| Объявление переменной | var <тип данных> <идентификатор>; |
| Присваивание | <идентификатор> = <значение>|<идентификатор>; |
| Объявление метода | <тип данных> method <идентификатор> (<тип данных> <идентификатор>, …) {…} |
| Блок инструкций | {  …  } |
| Возврат из подпрограммы | recive <выражение> |
| Вывод данных | write <выражение> |
| Вызов функций | <идентификатор функции>(<идентификатор | литерал>,…) |

## **Операции языка**

## Язык программирования SKG-2023 может выполнять арифметические операции, представленные в таблице 1.7.

Таблица 1.7 – Операции и их приоритеты

|  |  |
| --- | --- |
| Операция | Приоритет операции |
| (  ) | 0 |
| , | 1 |
| +  - | 2 |
| \*  / (деление нацело)  ? (деление по модулю) | 3 |

## **Выражения и их вычисления**

Круглые скобки в выражении используются для изменения приоритета операций. Также не допускается запись двух подряд идущих арифметических операций. Выражение может содержать вызов функции.

## **Программные конструкции языка**

Ключевые программные конструкции языка программирования SKG-2023 представлены в таблице 1.8

Таблица 1.8 – программные конструкции

|  |  |
| --- | --- |
| Главная функция (точка входа в приложение) | Primary  {…} |
| Метод | method <тип данных> <идентификатор> (<тип данных> <идентификатор>, …)  {… recive <выражение>;} |
| Условная конструкция condition | condition(<условие>){…}  otherwise{…} |

## **Область видимости идентификаторов**

Перед использованием переменной необходимо её объявление. Допускается использование переменной только внутри её области видимости. Допускается объявление переменных с одинаковыми именами в разных программных блоках.

## **Семантические проверки**

Таблица с перечнем семантических проверок, предусмотренных языком, приведена в таблице 1.8.

Таблица 1.8 – Семантические проверки

|  |  |
| --- | --- |
| Номер | Правило |
| 1 | Идентификаторы функций не должны повторяться |
| 2 | Операнды в операторах ветвления и выхода из функции должны быть целочисленного или логического типа |
| 3 | Тип данных передаваемых значений в функцию должен совпадать с типом параметров при её объявлении |
| 4 | Тип данных передаваемых значений в функцию стандартной библиотеки должен соответствовать заявленному. |
| 5 | Идентификатор должен быть объявлен до его использования. |

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

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

## **Стандартная библиотека и её состав**

Функции стандартной библиотеки с описанием представлены в таблице 1.9. Стандартная библиотека написана на языке программирования C++.

Таблица 1.9 – Состав стандартной библиотеки

|  |  |  |  |
| --- | --- | --- | --- |
| Имя функции | Возвращаемое значение | Принимаемые параметры | Описание |
| BREAKL | - | - | Переход на следующую строку |
| OutputInt | - | int a | Функция выводит в консоль число a |
| OutputChar | - | int a | Функция выводит в консоль символ a |
| OutputBoolean | - | int a | Функция выводит в консоль true(false)  в зависимости от параметра а - 1(0) |

## **Ввод и вывод данных**

В языке программирования SKG-2023 ввод данных не поддерживается. Вывод данных происходит с помощью функции write(<идентификатор>|<литерал>);

## **Точка входа**

Точкой входа является функция primary.

## **Препроцессор**

Препроцессор в языке программирования SKG-2023 не предусмотрен.

## **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**

SKG-2023 транслируется в язык ассемблера.

## **Классификация сообщений транслятора**

В случае возникновения ошибки в коде программы на языке SKG-2023 и выявления её транслятором в текущий файл протокола выводится сообщение. Их классификация сообщений приведена в таблице 1.10.

Таблица 1.10. Классификация сообщений транслятора

|  |  |
| --- | --- |
| Интервал | Описание ошибок |
| 0-110 | Системные ошибки, ошибки параметров |
| 200-220 | Ошибки лексического анализа |
| 300-400 | Ошибки семантического анализа |
| 600-620 | Ошибки синтаксического анализа |

## **Контрольный пример**

Контрольный пример представлен в приложении А.

# **Глава 2. Структура транслятора**

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

Транслятор преобразует программу, написанную на языке SKG-2023 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

Лексический анализ – первая фаза трансляции. Назначением лексического анализатора является нахождение ошибок лексики языка SKG-2023 и формирование таблицы лексем и таблицы идентификаторов.

Синтаксический анализ – это основная часть транслятора, предназначенная для распознавания синтаксических конструкций и формирования промежуточного кода SKG-2023. Для этого используются таблица лексем и идентификаторов. Синтаксический анализатор распознаёт синтаксические конструкции, выявляет синтаксические ошибки при их наличии и формирует дерево разбора

Семантический анализ в свою очередь является проверкой исходной программы SKG-2023 на семантическую согласованность с определением языка, т.е. проверяет правильность текста исходной программы с точки зрения семантики.

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 SKG-2023, прошедший успешно все предыдущие этапы, в код на языке Ассемблера.

## **2.2 Перечень входных параметров транслятора**

Входные параметры представлены в таблице 2.1.

Таблица 2.1 – Входные параметры транслятора языка SKG-2023

|  |  |  |
| --- | --- | --- |
| Входной параметр | Описание | Значение по умолчанию |
| -in:<имя\_файла> | Входной файл с любым расширением, в котором содержится исходный код на языке SKG-2023. 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 SKG-2023. 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 |

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

## **2.3 Перечень протоколов, формируемых транслятором и их содержимое**

Таблица с перечнем протоколов, формируемых транслятором языка SKG-2023 и их назначением представлена в таблице 2.2.

Таблица 2.2 – Протоколы, формируемые транслятором языка SKG-2023

|  |  |
| --- | --- |
| Формируемый протокол | Описание протокола |
| Файл журнала, log | Файл содержит в себе краткую информацию об исходном коде на языке SKG-2022. В этот файл выводится различные ошибки, а также результат работы анализаторов |
| Файл журнала, logOut | Файл содержит в себе краткую информацию об исходном коде на языке SKG-2022. В этот файл выводится протокол работы анализаторов. |
| 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.

# **Глава 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 SVG-2023. На выходе формируется таблица лексем и таблица идентификаторов. Структура лексического анализатора представлена на рисунке 3.1.



Рисунок 3.1 Структура лексического анализатора SKG-2023

**3.2 Входные данные лексического анализатора**

Лексический анализатор преобразует исходный код программы в структурированный формат, который затем используется более высокоуровневыми компонентами компилятора для выполнения синтаксического анализа и последующих этапов компиляции.

## **3.3 Параметры лексического анализатора и режимы его работы**

Входным параметром лексического анализатора является структура IN, которая содержит исходный текст программы, написанный на языке SKG-2023, а также структура LOG, которая содержит файл протокола.

## **3.4 Алгоритм лексического анализа**

Лексический анализ выполняется программой (входящей в состав транслятора), называемой лексическим анализатором. Цель лексического анализа — выделение и классификация лексем в тексте исходной программы. Лексический анализатор производит распознаёт и разбирает цепочки исходного текста программы. Это основывается на работе конечных автоматов, которую можно представить в виде графов. 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. Для ускорения работы анализатора я добавил просмотр первого символа слова, за счет этого отсеиваются неподходящие графы. Результат работы лексического анализатора – сформированные таблицы лексем и идентификаторов.

Пример. Регулярное выражение для ключевого слова main.

Граф конечного автомата для этой лексемы представлен на рисунке 3.6. S0 – начальное состояние, S4 – конечное состояние автомата. В виде кода представлен на рисунке 3.7.

Изображение выглядит как Шрифт, линия

Автоматически созданное описание

Рисунок 3.6 — Граф переходов для цепочки “primary”

Изображение выглядит как текст, Шрифт, снимок экрана, число

Автоматически созданное описание

Рисунок 3.7 — Граф переходов для цепочки “ primary ”

## **3.5 Контроль входных символов**

Таблица для контроля входных символов представлена на рисунке 3.2

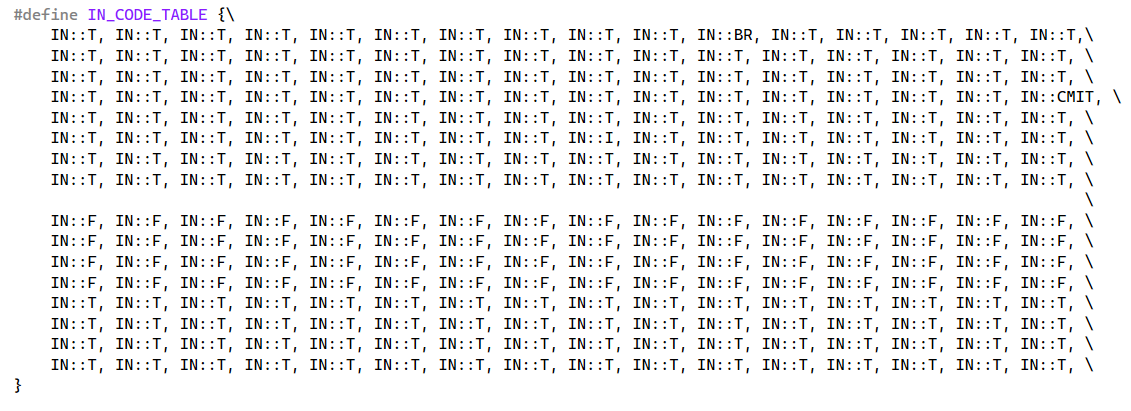


Рисунок 3.2. Таблица контроля входных символов

Принцип работы таблицы заключается в соответствии значения каждому элементу в шестнадцатеричной системе счисления значению в таблице WINDOWS-1251.

Описание значения символов: T – разрешённый символ, F – запрещённый символ, I – игнорируемый символ, BR – символ перехода на новую строку (\n), CMIT – символ комментария(?).

## **3.6 Удаление избыточных символов**

Избыточными символами являются символы табуляции и пробелы.

Избыточные символы удаляются на этапе разбиения исходного кода на лексемы.

Описание алгоритма удаления избыточных символов:

1. Посимвольно считываем исходный код, занесенный в структуру In.

2. Встреча пробела или знака табуляции вне пределов строкового литерала является своего рода встречей символа-сепаратора.

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

## **3.7 Перечень ключевых слов, сепараторов, символов операций и соответствующих им лексем, регулярных выражений и конечных автоматов**

Лексемы – это символы, соответствующие ключевым словам, символам операций и сепараторам, необходимые для упрощения дальнейшей обработки исходного кода программы. Данное соответствие описано в таблице 3.1.

Таблица 3.1 Соответствие ключевых слов, символов операций и сепараторов с лексемами

|  |  |  |  |
| --- | --- | --- | --- |
| Тип цепочки | Цепочка | | Лексема |
| Ключевые слова | var | | d |
| int, string, boolean, symbol | | t |
| primary | | m |
| method | | f |
| procedure | | p |
| recive | | r |
| write | | o |
| cycle | | u |
| condition | | w |
| otherwise | | ! |
| Иное | Идентификатор | | i |
| Литерал | | l |
| Сепараторы | ; | ; | |
| , | , | |
| { | { | |
| } | } | |
| ( | ( | |
| ) | ) | |
| Операторы | Арифметические (+, -, \*, /, %), Cравнения (!,~,<,>) | v | |
| Присваивание (=) | = | |

Пример реализации таблицы лексем представлен в приложении Б.

Также в приложении В находятся конечные автоматы, соответствующие лексемам языка SKG-2023.

## **3.8 Основные структуры данных**

Описание основных структур данных, используемых для хранения таблиц идентификаторов, представлено на рис. 3.3.

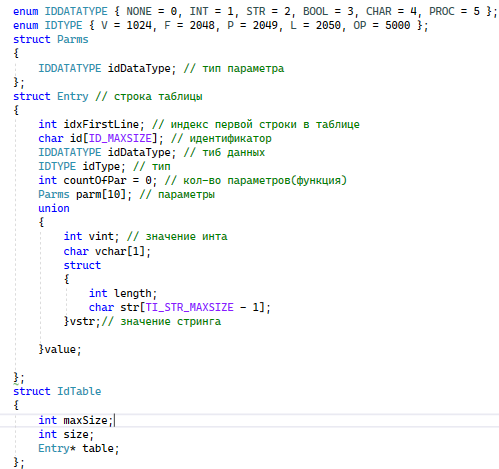


Рисунок 3.3. — Структуры таблиц идентификаторов SKG-2023

Описание основных структур данных, используемых для хранения таблиц лексем, представлено на рис. 3.4.

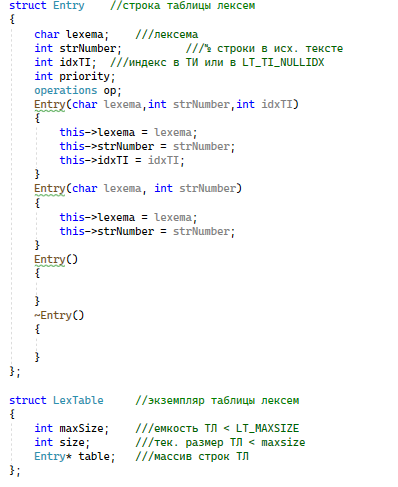


Рисунок 3.4 — Структуры таблиц лексем SKG-2023

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

Перечень сообщений лексического анализатора представлен на рисунке 3.5.

Изображение выглядит как текст, Шрифт, снимок экрана

Автоматически созданное описание

Рисунок 3.5 – Перечень ошибок лексического анализатора

## **3.10 Принцип обработки ошибок**

При обнаружении ошибки лексическим анализатором, он обращается к таблице, содержащей соответствия между номера ошибок и их сообщениями, и выводит в файл протокол ошибку (при возможности указывая место, приведшее к ошибке). При обнаружении ошибки дальнейшая работа транслятора прекращается, и он завершает работу.

## **3.11 Контрольный пример**

Результат работы лексического анализатора –сформированные таблицы лексем и идентификаторов – представлены в приложении Б.

# **Глава 4. Разработка синтаксического анализатора**

## **4.1 Структура синтаксического анализатора**

Синтаксический анализ – это фаза трансляции, выполняемая после лексического анализа и предназначенная для распознавания синтаксических конструкций. Входом для синтаксического анализа является таблица лексем и таблица идентификаторов, полученные после фазы лексического анализа. Выходом – дерево разбора. Структура синтаксического анализатора представлена на рисунке 4.1.



Рисунок 4.1 – Структура синтаксического анализатора

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

В синтаксическом анализаторе транслятора языка SKG-2023 используется контекстно-свободная грамматика , где

T – множество терминальных символов (представлены в приложении З)

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.

Правила языка SKG-2023 представлена в приложении Г.

TS – терминальные символы, которыми являются сепараторы, знаки арифметических операций и некоторые строчные буквы.

NS – нетерминальные символы, представленные несколькими заглавными буквами латинского алфавита.

Таблица 4.1 – Перечень правил, составляющих грамматику языка и описание нетерминальных символов SKG-2023

|  |  |  |
| --- | --- | --- |
| Нетерминал | Цепочки правил | Описание |
| S | ftiFBS  piFUS  m{N}  ftiFB  piFU | Проверка правильности структуры программы |
| F | (P)  () | Проверка наличия списка параметров функции |
| P | ti  ti,P | Проверка на ошибку в параметрах функции при ее объявлении |
| B | {NrI;}  {rI;} | Проверка наличия тела функции |
| I | l  i | Проверка на недопустимое выражение (ожидается только литерал или идентификатор) |
| N | dti;N  dti;  dti=E;N  dti=E;  i=E;N  i=E;  u(R){L}N  u(R){ L }  w(R){ L }N  w(R){ L }  w(R){ L }!{ L }N  w(R){ L }!{ L }  o[I];N  o[I];  b;N  b;  iK;N  iK; | Проверка на неверную конструкцию в теле функции |

Продолжение таблицы 4.1

|  |  |  |
| --- | --- | --- |
| K | (W)  () | Проверка на ошибку в вызове функции |
| E | i  iM  l  lM  (E)  (E)M  iK  iKM | Проверка на ошибку в арифметическом выражении |
| W | i  l  i,W  l,W | Проверка на ошибку в параметрах вызываемой функции |
| M | vE  vEM | Проверка арифметических действий |
| L | i=E;N  i=E;  oi;N  oK;  b;N  b;  iK;N  iK; | Проверка на неверную конструкцию в теле цикла или условного выражения |

## **4.3 Построение конечного магазинного автомата**

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

Таблица 4.2 – Описание компонентов магазинного автомата

|  |  |  |
| --- | --- | --- |
| Компонента | Определение | Описание |
|  | Множество состояний автомата | Состояние автомата представляет из себя структуру, содержащую позицию на входной ленте, номера текущего правила и цепочки и стек автомата |
|  | Алфавит входных символов | Алфавит является множеством терминальных и нетерминальных символов, описание которых содержится в разделе 1.2 и в таблице 4.1. |

Продолжение таблицы 4.2

|  |  |  |
| --- | --- | --- |
|  | Алфавит специальных магазинных символов | Алфавит магазинных символов содержит стартовый символ и маркер дна стека |
|  | Функция переходов автомата | Функция представляет из себя множество правил грамматики, описанных в таблице 4.1. |
|  | Начальное состояние автомата | Состояние, которое приобретает автомат в начале своей работы. Представляется в виде стартового правила грамматики (нетерминальный символ 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 |

## **4.4 Основные структуры данных**

Основные структуры данных синтаксического анализатора включают в себя структуру магазинного автомата и структуру грамматики Грейбах, описывающей правила языка SKG-2023. Данные структуры представлены в приложении Д.

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

Входным параметром синтаксического анализатора является структура LEX, которая содержит сформированную таблицу лексем, полученную на этапе лексического анализа, потоки вывода протокола, а также правила контекстно-свободной грамматики в форме Грейбах.

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

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

Перечень сообщений синтаксического анализатора представлен на рисунке 4.1.

Изображение выглядит как текст, снимок экрана, Шрифт

Автоматически созданное описание

Рисунок 4.1 – Перечень сообщений синтаксического анализатора

## **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. Синтаксический анализатор работает до 3 ошибок.
4. В случае нахождения ошибки, после всей процедуры трассировки в протокол будет выведено диагностическое сообщение.

## **4.9 Контрольный пример**

Пример разбора синтаксическим анализатором исходного кода предоставлен в приложении Е в виде фрагмента трассировки и дерева разбора исходного кода.

# **Глава 5. Разработка семантического анализатора**

## **5.1 Структура семантического анализатора**

Семантический анализ в трансляторе языка SKG-2023 выделен в отдельную фазу, и реализуется в виде отдельных проверок текущих ситуаций в конкретных случаях: установки флага или нахождении в особом месте программы (оператор выхода из функции, оператор ветвления, вызов функции стандартной библиотеки и т.д.).

## **5.2 Функции семантического анализатора**

За семантический анализ отвечает функция SemAnalyze. Ее входными параметрами является структура LEX, которая содержит таблицу лексем, идентификаторов и поток вывода в протокол.

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 5.1.

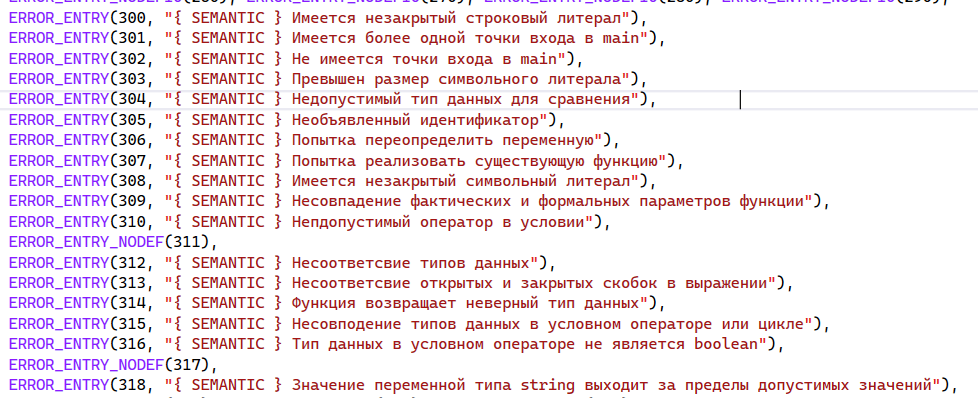


Рисунок 5.1 – Перечень сообщений семантического анализатора

## **5.4 Принцип обработки ошибок**

Семантический анализатор - проверяет, что объявления и утверждения программы семантически верны. Например: соответствие типов данных в выражении, совпадение фактических и формальных параметров функции.

Обработка ошибок происходит следующим образом:

1. Анализатор перебирает таблицу лексем.
2. Проверяет исключительные ситуации, которые могут быть связаны с данной лексемой.
3. В случае нахождения ошибки, она записывается в log журнал и дублируется в консоль
4. По завершению работы в log журнал печатается сообщение об успешной или ошибочной работе анализатора.

## **5.5 Контрольный пример**

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

Таблица 5.3 – Тестирование функций

|  |  |
| --- | --- |
| Исходный код с ошибкой | Генерируемое сообщение об ошибке |
| {  var int check = 5;  write check;  } | ERROR CODE 302: { SEMANTIC } Не имеется точки входа в primary  Строка -1 позиция -1 |
| main  {  var int check = 5;  write check;  write "hello;  } | ERROR CODE 300: { SEMANTIC } Имеется незакрытый строковый литерал  Строка -1 позиция -1 |
| main  {  var int check = 5;  var int check = 2;  } | ERROR CODE 306, { SEMANTIC } Попытка переопределить переменную  Строка n позиция -1 |

# **Глава 6. Преобразование выражений**

## **6.1 Выражения, допускаемые языком**

В языке SKG-2023 допускаются выражения, применимые к целочисленным типам данных. В выражениях поддерживаются арифметические операции, такие как +, -, \*, /, %(остаток от деления) и (), и вызовы функций как операнды арифметических выражений.

Приоритет операций представлен в таблице 6.1.

Таблица 6.1 – Приоритет операций в языке SKG-2023

|  |  |
| --- | --- |
| Приоритет | Операция |
| 0 | ( ) |
| 1 | , |
| 2 | + - |
| 3 | \* / % |

## **6.2 Польская запись**

Выражения в языке SKG-2023 преобразовываются к обратной польской записи.

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

Обратная польская запись — это форма записи математических выражений, в которой операторы расположены после своих операндов. Выражение в обратной польской нотации читается слева направо: операция выполняется над двумя операндами, непосредственно стоящими перед знаком этой операции.

Алгоритм построения:

– исходная строка: выражение;

– результирующая строка: польская запись;

– стек: пустой;

– результирующая строка: польская запись;

– исходная строка просматривается слева направо;

– операнды переносятся в результирующую строку в порядке их следования;

– операция записывается в стек, если стек пуст или в вершине стека лежит отрывающая скобка;

– операция выталкивает все операции с большим или равным приоритетом в результирующую строку;

– запятая не помещается в стек, если в стеке операции, то все выбираются в строку;

– отрывающая скобка помещается в стек;

– закрывающая скобка выталкивает все операции до открывающей скобки, после чего обе скобки уничтожаются;

– закрывающая скобка с приоритетом, равным 4, выталкивает все до открывающей с таким же приоритетом и генерирует @ – специальный символ, в которого записывается информация о вызываемой функции, а в поле приоритета для данной лексемы записывается число параметров вызываемой функции;

– по концу разбора исходной строки все операции, оставшиеся в стеке, выталкиваются в результирующую строку.

## **6.3 Программная реализация обработки выражений**

Программная реализация алгоритма преобразования выражений в обратный польский формат основана функциях PolishNotation и StartPolish. Функция StartPolish принимает как параметр адрес таблицы лексем и содержит цикл, в ходе которого перебираются все лексемы исходного кода. Если последовательность лексем соответствует началу выражения, вызывается функция PolishNotation, где и проводится точечное преобразование выражений к польской нотации.

## **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.

В приложении Ж приведен измененное представление промежуточного кода, отображающее результаты преобразования выражений в польский формат.

Таблица 6.2 – Пример преобразования выражения в обратную польскую запись

|  |  |  |
| --- | --- | --- |
| Исходная строка | Результирующая строка | Стек |
| q\*2 - s(i) |  |  |
| \*2 - s(i) | q |  |
| 2 - s(i) | q | \* |
| - s(i) | q2 | \* |
| s(i) | q2\* | - |
| (i) | q2\* | - |
| i) | q2\* | - |
| ) | q2\*i | - |
|  | q2\*i@1- |  |

# **Глава 7. Генерация кода**

## **7.1 Структура генератора кода**

Генерация объектного кода — это перевод компилятором внутреннего представления исходной программы в цепочку символов выходного языка. На вход генератора подаются таблицы лексем и идентификаторов, на основе которых генерируется файл с ассемблерным кодом.

Структура генератора кода представлена на рисунке 7.1.



Рисунок 7.1 Структура генератора кода

Функция генератора кода представлена на рисунке 7.2.

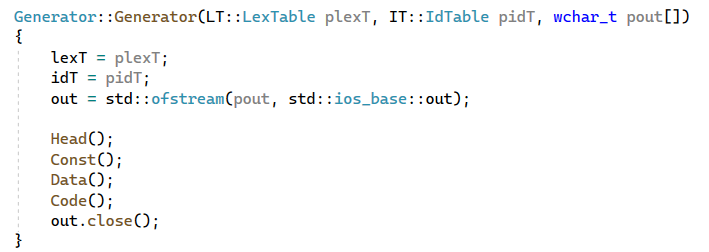


Рисунок 7.2 – Функция генератора кода

Функция Head отвечает за установку необходимого соглашения о вызовах, модели памяти, Подключение библиотек и объявление прототипов статической библиотеки.

Функция Const отвечает за создание сегмента констант, куда заносятся лексемы имеющихся типов данных. Циклом проходим по таблицы идентификаторов, если находим лексему, то смотрим ее тип данных и даем ей эквивалентный тип данных в ассемблере.

Функция Data отвечает за создание сегмента данных, куда заносятся идентификаторы имеющихся типов данных.

Функция Code отвечает за создание сегмента кода, инструкции, написанные на языке SKG-2023 генерируются в Assembler.

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

Элементы таблицы идентификаторов расположены в разных сегментах языка ассемблера – .data и .const. Идентификаторы языка SKG-2023 размещены в сегменте данных(.data). Литералы – в сегменте констант (.const). Соответствия между типами данных идентификаторов на языке SKG-2023 и на языке ассемблера приведены в таблице 7.1. Сгенерированный код предоставлен в приложении И.

Таблица 7.1 – Соответствия типов идентификаторов языка SKG-2023 и языка Ассемблера

|  |  |  |
| --- | --- | --- |
| Тип идентификатора на языке SAV-2022 | Тип идентификатора на языке ассемблера | Пояснение |
| int | SDWORD | Хранит целочисленный тип данных со знаком. 4 байта. |
| symbol | word | Символ хранится размером в 2 байт. |
| boolean | word | Переменная хранится размером в 2 байт. |

## **7.3 Статическая библиотека**

Функции из стандартной библиотеки содержатся в проекте StaticLibrary, в свойствах которого указан тип конфигурации «статическая библиотека». Подключение библиотеки происходит с помощью includelib на этапе генерации кода путем вывода в поток out. Таким же образом c помощью оператора EXTRN объявляются названия функций из библиотеки. Оператор EXTRN выполняет две функции. Во-первых, он сообщает ассемблеру, что указанное символическое имя является внешним для текущего ассемблирования. Вторая функция оператора EXTRN состоит в том, что он указывает ассемблеру тип соответствующего символического имени.

**7.4 Особенности алгоритма генерации кода**

Так как ассемблирование является очень формальной процедурой, то ассемблер должен знать, что представляет из себя каждый символ. Это позволяет ему генерировать правильные команды. Вышеописанное проиллюстрировано на рисунке 7.3.

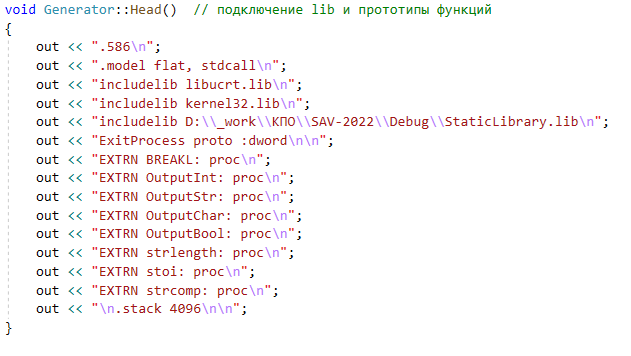


Рисунок 7.3 Фрагмент функции генерации кода

**7.5 Параметры управляющие генерацией кода**

Параметры, управляющие генерацией кода, транслятором не предусмотрены.

**7.6 Контрольный пример**

В приложении К представлен код, преобразованный транслятором из языка SKG-2023 в язык Ассемблера.

# **Глава 8. Тестирование транслятора**

## **8.1Тестирование 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 |
| var symbol g = 'f; | ERROR CODE 308: { SEMANTIC } Имеется незакрытый символьный литерал  Строка 1 позиция -1 |

## **8.2 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 |
| int method kek( int z)  {  var int n = z+5;  } | ERROR CODE 605: { SYNTACTIC } неверная структура метода  Строка 4 позиция -1 |

## **8.3 Тестирование семантического анализатора**

Итоги тестирования семантического анализатора приведены в пункте 5.5.

**Заключение**

По окончании выполнения поставленных задач был получен транслятор языка SKG-2023 на язык Ассемблера.

Транслятор поддерживает 3 типа данных: «integer», «char», «bool».

Транслятор поддерживает объявления переменных, присваивания, условные конструкции «if», определение и вызов функций.

Транслятор поддерживает 5 арифметических операций и скобки «+», «-», «\*», «/», «%», «(»,.

Транслятор поддерживает функцию стандартной библиотеки «rand()».

Транслятор проверяет исходный код 16 семантическими проверками.

**Приложения**

## **Приложение А**

boolean method EvenCheck(int k)

{

var boolean h;

k = k % 2;

condition (k ~ 0)

{

h = true;

}

otherwise

{

h = false;

}

recive h;

}

int method r(int s){

s = (s\* 214013 + 2531011) % 2147483648;

recive s;

}

primary

{

var boolean iseven = EvenCheck(10);

write iseven;

hortab;

var symbol q='6';

write q;

var int l=6/2;

write l;

var int z=r(1);

write z;

}

## **Приложение Б**

ID Table

---------------------------------------------------------------------------------------

№ | Identifier | Data Type | Identifier Type | Index in LT | Value

---------------------------------------------------------------------------------------

0000 | EvenCheck | boolean | метод | 2 | -

0001 | EvenCheckk | integer | параметр | 5 | -

0002 | EvenCheckh | boolean | переменная | 10 | 0

0003 | L1 | integer | литерал | 16 | 2

0004 | L2 | integer | литерал | 22 | 0

0005 | L3 | boolean | литерал | 27 | 1

0006 | L4 | boolean | литерал | 34 | 0

0007 | r | integer | метод | 43 | -

0008 | rs | integer | параметр | 46 | -

0009 | L5 | integer | литерал | 54 | 214013

0010 | L6 | integer | литерал | 56 | 2531011

0011 | L7 | integer | литерал | 59 | 2147483647

0012 | primary | integer | метод | 65 | -

0013 | primaryiseven | boolean | переменная | 69 | 0

0014 | L8 | integer | литерал | 73 | 10

0015 | primaryq | symbol | переменная | 83 | ''

0016 | L9 | symbol | литерал | 85 | '6'

0017 | primaryl | integer | переменная | 89 | 0

0018 | L10 | integer | литерал | 91 | 6

0019 | primaryz | integer | переменная | 104 | 0

0020 | L11 | integer | литерал | 108 | 1

---------------------------------------------------------------------------------------

Количество идентификаторов: 21

---------------------------------------------------------------------------------------

Начало таблицы лексем:

Lexical Table

----------------------------------------

№ | lex | idx in IT |номер строки |

----------------------------------------

0000 | t | | 1 |

0001 | f | | 1 |

0002 | i | 0 | 1 |

0003 | ( | | 1 |

0004 | t | | 1 |

0005 | i | 1 | 1 |

0006 | ) | | 1 |

0007 | { | | 2 |

0008 | d | | 3 |

0009 | t | | 3 |

0010 | i | 2 | 3 |

0011 | ; | | 3 |

0012 | i | 1 | 4 |

0013 | = | | 4 |

0014 | i | 1 | 4 |

0015 | v | 3 | 4 |

0016 | l | 4 | 4 |

0017 | ; | | 4 |

Окончание таблицы лексем:

0111 | o | | 36 |

0112 | i | 25 | 36 |

0113 | ; | | 36 |

0114 | } | | 42 |

ПРОМЕЖУТОЧНОЕ ПРЕДСТАВЛЕНИЕ КОДА:

0001| tfi(ti){dti;

0002| i=ilv;

0003| w(ivl){i=l;

0004| }!{i=l;

0005| }ri;

0006| }tfi(ti){i=ilvlvlv##;

0007| ri;

0008| }m{dti=l@1#;

0009| oi;

0010| b;

0011| dti=l;

0012| dti=lllvv##;

0013| oi;

0014| dti=l@1#;

0015| oi;

0016| }

## **Приложение В**

## define FST\_DECLARE 4, \

FST::NODE(1, FST::RELATION('v', 1)),\

FST::NODE(1, FST::RELATION('a', 2)),\

FST::NODE(1, FST::RELATION('r', 3)),\

FST::NODE()

#define FST\_INTEGER 4, \

FST::NODE(1, FST::RELATION('i', 1)),\

FST::NODE(1, FST::RELATION('n', 2)),\

FST::NODE(1, FST::RELATION('t', 3)),\

FST::NODE()

#define FST\_BOOL 8, \

FST::NODE(1, FST::RELATION('b', 1)),\

FST::NODE(1, FST::RELATION('o', 2)),\

FST::NODE(1, FST::RELATION('o', 3)),\

FST::NODE(1, FST::RELATION('l', 4)),\

FST::NODE(1, FST::RELATION('e', 5)),\

FST::NODE(1, FST::RELATION('a', 6)),\

FST::NODE(1, FST::RELATION('n', 7)),\

FST::NODE()

#define FST\_CHAR 7, \

FST::NODE(1, FST::RELATION('s', 1)),\

FST::NODE(1, FST::RELATION('y', 2)),\

FST::NODE(1, FST::RELATION('m', 3)),\

FST::NODE(1, FST::RELATION('b', 4)),\

FST::NODE(1, FST::RELATION('o', 5)),\

FST::NODE(1, FST::RELATION('l', 6)),\

FST::NODE()

#define FST\_STRING 7, \

FST::NODE(1, FST::RELATION('s', 1)),\

FST::NODE(1, FST::RELATION('t', 2)),\

FST::NODE(1, FST::RELATION('r', 3)),\

FST::NODE(1, FST::RELATION('i', 4)),\

FST::NODE(1, FST::RELATION('n', 5)),\

FST::NODE(1, FST::RELATION('g', 6)),\

FST::NODE()

#define FST\_FUNCTION 7, \

FST::NODE(1, FST::RELATION('m', 1)),\

FST::NODE(1, FST::RELATION('e', 2)),\

FST::NODE(1, FST::RELATION('t', 3)),\

FST::NODE(1, FST::RELATION('h', 4)),\

FST::NODE(1, FST::RELATION('o', 5)),\

FST::NODE(1, FST::RELATION('d', 6)),\

FST::NODE()

#define FST\_BREAKL 7, \

FST::NODE(1, FST::RELATION('h', 1)),\

FST::NODE(1, FST::RELATION('o', 2)),\

FST::NODE(1, FST::RELATION('r', 3)),\

FST::NODE(1, FST::RELATION('t', 4)),\

FST::NODE(1, FST::RELATION('a', 5)),\

FST::NODE(1, FST::RELATION('b', 6)),\

FST::NODE()

## **Приложение Г**

Greibach greibach(

NS('S'), TS('$'),

13,

Rule(

NS('S'), GRB\_ERROR\_SERIES + 0,// неверная структура программы

5,

Rule::Chain(6, TS('t'), TS('f'), TS('i'), NS('F'), NS('B'), NS('S')),

Rule::Chain(5, TS('p'), TS('i'), NS('F'), NS('U'), NS('S')),

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

Rule::Chain(5, TS('t'), TS('f'), TS('i'), NS('F'), NS('B')),

Rule::Chain(4, TS('p'), TS('i'), NS('F'), NS('U'))

),

Rule(

NS('N'), GRB\_ERROR\_SERIES + 1, // неверная структура фунцкии

18,

Rule::Chain(5, TS('d'), TS('t'), TS('i'), TS(';'), NS('N')),

Rule::Chain(7, TS('d'), TS('t'), TS('i'), TS('='), NS('E'), TS(';'), NS('N')),

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

Rule::Chain(8, TS('u'), TS('('), NS('C'), TS(')'), TS('{'), NS('L'), TS('}'), NS('N')),

Rule::Chain(8, TS('w'), TS('('), NS('C'), TS(')'), TS('{'), NS('L'), TS('}'), NS('N')),

Rule::Chain(12, TS('w'), TS('('), NS('C'), TS(')'), TS('{'), NS('L'), TS('}'), TS('!'), TS('{'), NS('L'), TS('}'), NS('N')),

Rule::Chain(4, TS('o'), NS('I'), TS(';'), NS('N')),

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

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

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

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

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

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

Rule::Chain(7, TS('u'), TS('('), NS('C'), TS(')'), TS('{'), NS('L'), TS('}')),

Rule::Chain(7, TS('w'), TS('('), NS('C'), TS(')'), TS('{'), NS('L'), TS('}')),

Rule::Chain(11, TS('w'), TS('('), NS('C'), TS(')'), TS('{'), NS('L'), TS('}'), TS('!'), TS('{'), NS('L'), TS('}')),

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

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

),

Rule(

NS('E'), GRB\_ERROR\_SERIES + 2, // ошибка в выражении

8,

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

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

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

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

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(3, TS('i'), NS('K'), NS('M'))

),

Rule(

NS('F'), GRB\_ERROR\_SERIES + 3,// ошибка объявления функции или процедуры

2,

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

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

),

Rule(

NS('P'), GRB\_ERROR\_SERIES + 4, // ошибка в параметрах функции или процедуры

2,

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

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

),

Rule(

NS('B'), GRB\_ERROR\_SERIES + 5, // неверная структура метода

2,

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

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

),

Rule(

NS('U'), GRB\_ERROR\_SERIES + 6, // неверная структура процедуры

1,

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

),

Rule(

NS('W'), GRB\_ERROR\_SERIES + 7, // ошибка в параметрах вызываемой функции

4,

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

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

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

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

),

Rule(

NS('C'), GRB\_ERROR\_SERIES + 8,// неверное условие цикла или уловного оператора C - нетерменал условия

4,

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

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

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

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

),

Rule(

NS('K'), GRB\_ERROR\_SERIES + 9,// ошибка при вызове функции

2,

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

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

),

Rule(

NS('M'), GRB\_ERROR\_SERIES + 10, // ошибка в операторе

2,

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

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

),

Rule(

NS('L'), GRB\_ERROR\_SERIES + 11, // ошибка в теле цикла

9,

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

Rule::Chain(4, TS('o'), NS('I'), TS(';'), NS('N')),

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

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

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

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

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

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

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

),

Rule(

NS('I'), GRB\_ERROR\_SERIES + 20,

2,

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

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

)

);

## **Приложение Д**

#pragma once

#include "Greibach.h"

#include <stack>

#include "LexAnalysis.h"

#include <queue>

#include<iostream>

#define MFST\_DIAGN\_MAXSIZE 2\*ERROR\_MAXSIZE\_MESSAGE

#define MFST\_DIAGN\_NUMBER 3

class my\_stack\_SHORT :public std::stack<short> {

public:

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

};

static int FST\_TRACE\_n = -1;

static char rbuf[205], sbuf[205], lbuf[1024];

#define MFST\_TRACE\_START out<< std::setw(4)<<std::left<<"Шаг"<<":"\

<< std::setw(20)<<std::left<<" Правило"\

<< std::setw(30)<<std::left<<" Входная лента"\

<< std::setw(20)<<std::left<<" Стек"\

<< std::endl;

#define MFST\_TRACE1 out<< std::setw(4)<<std::left<<++FST\_TRACE\_n<<": "\

<< std::setw(20)<<std::left<<rule.GetCRule(rbuf,nruleChain)\

<< std::setw(30)<<std::left<<GetCLenta(lbuf,lentaPosition)\

<< std::setw(20)<<std::left<<GetCSt(sbuf)\

<< std::endl;

#define MFST\_TRACE2 out<< std::setw(4)<<std::left<<FST\_TRACE\_n<<": "\

<< std::setw(20)<<std::left<<" "\

<< std::setw(30)<<std::left<<GetCLenta(lbuf,lentaPosition)\

<< std::setw(20)<<std::left<<GetCSt(sbuf)\

<< std::endl;

#define MFST\_TRACE3 out<< std::setw(4)<<std::left<<++FST\_TRACE\_n<<": "\

<< std::setw(20)<<std::left<<" "\

<< std::setw(30)<<std::left<<GetCLenta(lbuf,lentaPosition)\

<< std::setw(20)<<std::left<<GetCSt(sbuf)\

<< std::endl;

#define MFST\_TRACE4(c) out<<std::setw(4)<<std::left << ++FST\_TRACE\_n << ": "<<std::setw(20)<< std::left <<c<<std::endl;

#define MFST\_TRACE5(c) out<<std::setw(4)<<std::left << FST\_TRACE\_n << ": "<<std::setw(20)<< std::left <<c<<std::endl;

#define MFST\_TRACE6(c,k) out<<std::setw(4)<<std::left << FST\_TRACE\_n << ": "<<std::setw(20)<< std::left << c << k <<std::endl;

#define MFST\_TRACE7 out<<std::setw(4)<<std::left << state.lentaPosition << ": "\

<<std::setw(20)<< std::left << rule.GetCRule(rbuf,state.numRuleChain)\

<<std::endl;

typedef my\_stack\_SHORT MFSTSTSTACK; // стек автомата

namespace MFST

{

struct MfstState // состояние автомата (для сохранения)

{

short nrule;

short lentaPosition; // позиция на ленте

short numRuleChain; // номер текущей цепочки, текущего правила

MFSTSTSTACK stack; // стек автомата

MfstState();

MfstState

(

short position, // позиция на ленте

MFSTSTSTACK pst, // стек автомата

short pnRuleChain // номер текущей цепочки, текущего правила

);

MfstState(

short position, // позиция на ленте

short pnrule,

short pnRuleChain // номер текущей цепочки, текущего правила

);

MfstState(short pposition, MFSTSTSTACK pst, short pnrule, short pnrulechain);

};

struct Mfst // магазинный автомат

{

enum RC\_STEP // код возврата функии step

{

NS\_OK, // найдено правило и цепочка, цепочка записана в стек

NS\_NORULE, // не найдено правило грамматики (ошибка грамматики)

NS\_NORULECHAIN, // не найдена подходящая цепочка правила (ошибка в исходном коде)

NS\_ERROR, // неизвестный нетерминальный символ грамматики

TS\_OK, // тек. символ ленты == вершине стека, продвинулась лента, pop стека

TS\_NOK, // тек. символ ленты != вершине стека, восстановленно состояние

LENTA\_END,// текущая позиция ленты >= lentaSize

SURPRISE// неожиданный код возврата(ошибка в step)

};

struct MfstDiagnosis // диагностика

{

short lentaPosition; // позиция в ленте

RC\_STEP rcStep; // код завершения шага

short nrule; // номер правила

short nruleChain; // номер цепочки правила

MfstDiagnosis();

MfstDiagnosis( // диагностика

short plentaPosition, // позиция на ленте

RC\_STEP rcStep, // код завершения шага

short pnrule, // номер правила

short pnruleChain // номер цепочки правила

);

} diagnosis[MFST\_DIAGN\_NUMBER]; // последние самые глубокие сообщения

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

public:

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

};

GRABALPHABET\* lenta; // перекодированая (TS/NS) лента (из LEX)

short lentaPosition; // текущая позиция на ленте

short nrule; // номер текущего правила

short nruleChain; // номер текущей цепочки, текущего правила

short lentaSize; // размер ленты

GRB::Greibach greibach; // грамматика Грейбах

Lexis::LEX lex; // результат работы лексического анализатора

MFSTSTSTACK st; // стек автомата

//std::stack<MfstState> storeState; // стек для сохранения состояний

my\_stack\_MfstState storeState;

Mfst();

Mfst(

Lexis::LEX plex, // результат работы лексического анализатора

GRB::Greibach pgrebach // грамматика Грейбах

);

char\* GetCSt(char\* buf); // получить содержимое стека

char\* GetCLenta(char\* buf, short pos, short n = 25); // лента: n символов c pos

char\* GetDiagnosis(short n, char\* buf); // получить n-ую строку диагностики или 0x00

bool SaveState(std::ostream& out); // сохранить состояние автомата

bool ResetState(std::ostream& out); // восстановить состояние автомата

bool PushChain(

GRB::Rule::Chain chain // цепочка правила

);

RC\_STEP step(std::ostream& out); // выполнить шаг автомата

bool Start(std::ostream& out, std::ostream& outlog); // запустить автомат

bool SaveDiagnosis(

RC\_STEP pprcStep // код завершения шага

);

void PrintRules(std::ostream& out); // вывести последовательность правил

struct Deducation // вывод

{

short size; // количество шагов в выводе

short\* nrules; // номера правил грамматики

short\* nruleChains; // номера цепочек правил грамматики

Deducation() { size = 0; nrules = 0; nruleChains = 0; };

} deducation;

bool SaveDeducation(); // сохранить дерево вывода

};

void SyntacticAnalysis(Lexis::LEX lex, Log::LOG log, std::ostream& out);

## **Приложение Е**

Начало разбора

Трассировка:

Шаг : Правило Входная лента Стек

0 : S->tfiFBS S$

0 : SAVESTATE: 1

0 : tfiFBS$

1 : fi(ti){dti;i=ivl;w(ivl){i fiFBS$

2 : i(ti){dti;i=ivl;w(ivl){i= iFBS$

3 : (ti){dti;i=ivl;w(ivl){i=l FBS$

4 : F->(P) (ti){dti;i=ivl;w(ivl){i=l FBS$

Конец разбора

739 : I->i i;}}}}}}}}}}}}}}}}}}}}}}} I;}$

739 : SAVESTATE: 55

739 : i;}}}}}}}}}}}}}}}}}}}}}}} i;}$

740 : ;}}}}}}}}}}}}}}}}}}}}}}}} ;}$

741 : }}}}}}}}}}}}}}}}}}}}}}}}} }$

742 : }}}}}}}}}}}}}}}}}}}}}}}}} $

743 : LENTA\_END

принятые правила

0 : S->tfiFBS

3 : F->(P)

4 : P->ti

7 : B->{NrI;}

8 : N->dti;N

12 : N->i=E;N

14 : E->iM

15 : M->vE

16 : E->l

18 : N->w(C){L}!{L}

20 : C->ivl

25 : L->i=E;

27 : E->l

32 : L->i=E;

34 : E->l

38 : I->i

41 : S->tfiFBS

44 : F->(P)

45 : P->ti

48 : B->{NrI;}

49 : N->i=E;

51 : E->(E)M

52 : E->iM

53 : M->vE

54 : E->lM

55 : M->vE

56 : E->l

58 : M->vE

59 : E->l

62 : I->i

65 : S->m{N}

67 : N->dti=E;N

71 : E->iK

72 : K->(W)

73 : W->l

76 : N->oI;N

77 : I->i

79 : N->b;N

81 : N->dti=E;N

85 : E->l

87 : N->dti=E;N

91 : E->lM

92 : M->vE

93 : E->(E)

94 : E->lM

95 : M->vE

96 : E->l

99 : N->oI;N

100 : I->i

102 : N->dti=E;N

106 : E->iK

107 : K->(W)

108 : W->l

111 : N->oI;

112 : I->i

## **Приложение Ж**

ПРОМЕЖУТОЧНОЕ ПРЕДСТАВЛЕНИЕ КОДА:

0001| tfi(ti){dti;

0002| i=ilv;

0003| w(ivl){i=l;

0004| }!{i=l;

0005| }ri;

0006| }tfi(ti){i=ilvlvlv##;

0007| ri;

0008| }m{dti=l@1#;

0009| oi;

0010| b;

0011| dti=l;

0012| dti=lllvv##;

0013| oi;

0014| dti=l@1#;

0015| oi;

0016| }

**Приложение З**

.586

.model flat, stdcall

includelib libucrt.lib

includelib kernel32.lib

includelib C:\Users\givi\Documents\Курсовой проект КПО\StaticLibrary\StaticLiblary\StaticLibrary.lib

ExitProcess proto :dword

EXTRN BREAKL: proc

EXTRN OutputInt: proc

EXTRN OutputChar: proc

EXTRN OutputBool: proc

EXTRN stoi: proc

EXTRN strcomp: proc

.stack 4096

.const

L1 sdword 2

L2 sdword 0

L3 sdword 1

L4 sdword 0

L5 sdword 214013

L6 sdword 2531011

L7 sdword 2147483647

L8 sdword 10

L9 word '6'

L10 sdword 6

L11 sdword 1

.data

buffer BYTE 256 dup(0)

EvenCheckh word 0

primaryiseven word 0

primaryq word ?

primaryl sdword 0

primaryz sdword 0

.code

EvenCheck proc EvenCheckk : sdword

push EvenCheckk

push L1

pop ebx

pop eax

cdq

idiv ebx

push eax

pop EvenCheckk

mov eax, EvenCheckk

cmp eax, L2

jz m0

jnz m1

je m1

m0:

push L3

pop EvenCheckh

jmp e0

m1:

push L4

pop EvenCheckh

e0:

push EvenCheckh

jmp local0

local0:

pop eax

ret

EvenCheck ENDP

r proc rs : sdword

push rs

push L5

pop eax

pop ebx

mul ebx

push eax

push L6

pop eax

pop ebx

add eax, ebx

push eax

push L7

pop ebx

pop eax

cdq

idiv ebx

push edx

pop rs

push rs

jmp local1

local1:

pop eax

ret

r ENDP

main proc

push L8

pop edx

push L8

call EvenCheck

push eax

pop primaryiseven

push primaryiseven

call OutputBool

call BREAKL

push L9

pop primaryq

push L10

push L1

push L1

pop ebx

pop eax

sub eax, ebx

push eax

pop ebx

pop eax

cdq

idiv ebx

push eax

pop primaryl

push primaryl

call OutputInt

push L11

pop edx

push L11

call r

push eax

pop primaryz

push primaryz

call OutputInt

call ExitProcess

main ENDP

end main

# **Литература**

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. Ирвин К. Р. Язык ассемблера для процессоров Intel / К. Р. Ирвин. – M.: Вильямс, 2005. – 912с.
4. Герберт, Ш. Справочник программиста по C/C++ / Шилдт Герберт. - 3-е изд. – Москва : Вильямс, 2003. - 429 с.