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 KAO-2019»

Выполнил студент Костюкова Анна Олеговна

(Ф.И.О.)

Руководитель проекта ст.пр. Наркевич Аделина Сергеевна

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

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

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

Консультанты ст.пр. Наркевич Аделина Сергеевна

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

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

Нормоконтролер ст.пр. Наркевич Аделина Сергеевна

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

Курсовой проект защищен с оценкой

Минск 2019

Содержание

[**Введение** 4](#_Toc27710023)

[**1 СПЕЦИФИКАЦИЯ ЯЗЫКА ПРОГРАММИРОВАНИЯ** 5](#_Toc27710024)

[**1.1 Характеристика языка программирования** 5](#_Toc27710025)

[**1.2 Алфавит языка** 5](#_Toc27710026)

[**1.3 Применяемые сепараторы** 5](#_Toc27710027)

[**1.4 Применяемые кодировки** 5](#_Toc27710028)

[**1.5 Типы данных** 6](#_Toc27710029)

[**1.6 Преобразование типов данных** 7](#_Toc27710030)

[**1.7 Идентификаторы** 7](#_Toc27710031)

[**1.8 Литералы** 7](#_Toc27710032)

[**1.9 Объявление данных** 7](#_Toc27710033)

[**1.10 Инициализация данных** 8](#_Toc27710034)

[**1.11 Инструкции языка** 8](#_Toc27710035)

[**1.12 Операции языка** 9](#_Toc27710036)

[**1.13 Выражения и их вычисление** 9](#_Toc27710037)

[**1.14 Конструкции языка** 9](#_Toc27710038)

[**1.15 Область видимости идентификаторов** 10](#_Toc27710039)

[**1.16 Семантические проверки** 10](#_Toc27710040)

[**1.17 Распределение оперативной памяти на этапе выполнения** 11](#_Toc27710041)

[**1.18 Стандартная библиотека и ее состав** 11](#_Toc27710042)

[**1.19 Ввод и вывод данных** 11](#_Toc27710043)

[**1.20 Точка входа** 11](#_Toc27710044)

[**1.21 Препроцессор** 11](#_Toc27710045)

[**1.22 Соглашения о вызовах** 11](#_Toc27710046)

[**1.23 Объектный код** 12](#_Toc27710047)

[**1.24 Классификация сообщений транслятора** 12](#_Toc27710048)

[**1.25 Контрольный пример** 12](#_Toc27710049)

[**2 СТРУКТУРА ТРАНСЛЯТОРА** 13](#_Toc27710050)

[**2.1 Компоненты транслятора, их назначение и принципы взаимодействия** 13](#_Toc27710051)

[**2.2 Перечень входных параметров транслятора** 14](#_Toc27710052)

[**2.3 Перечень протоколов, формируемых транслятором и их содержимое** 14](#_Toc27710053)

[**3 РАЗРАБОТКА ЛЕКСИЧЕСКОГО АНАЛИЗАТОРА** 15](#_Toc27710054)

[**3.1 Структура лексического анализатора** 15](#_Toc27710055)

[**3.2 Контроль входных символов** 15](#_Toc27710056)

[**3.3 Удаление избыточных символов** 16](#_Toc27710057)

[**3.4 Перечень ключевых слов** 16](#_Toc27710058)

[**3.5 Основные структуры данных** 17](#_Toc27710059)

[**3.6 Принцип обработки ошибок** 17](#_Toc27710060)

[**3.7 Структура и перечень сообщений лексического анализатора** 17](#_Toc27710061)

[**3.8 Параметры лексического анализатора и режим его работы** 18](#_Toc27710062)

[**3.9 Алгоритм лексического анализа** 18](#_Toc27710063)

[**3.10 Контрольный пример** 18](#_Toc27710064)

[**4 РАЗРАБОТКА СИНТАКСИЧЕСКОГО АНАЛИЗАТОРА** 19](#_Toc27710065)

[**4.1 Структура синтаксического анализатора** 19](#_Toc27710066)

[**4.2 Контекстно свободная грамматика, описывающая синтаксис языка** 19](#_Toc27710067)

[**4.3 Построение конечного магазинного автомата** 20](#_Toc27710068)

[**4.4** **Основные структуры данных** 21](#_Toc27710069)

[**4.5** **Описание алгоритма синтаксического разбора** 21](#_Toc27710070)

[**4.6** **Структура и перечень сообщений синтаксического анализатора** 21](#_Toc27710071)

[**4.7** **Параметры синтаксического анализатора и режимы его работы** 22](#_Toc27710072)

[**4.8** **Принцип обработки ошибок** 22](#_Toc27710073)

[**4.9** **Контрольный пример** 22](#_Toc27710074)

[**5 РАЗРАБОТКА СЕМАНТИЧЕСКОГО АНАЛИЗАТОРА** 23](#_Toc27710075)

[**5.1 Структура семантического анализатора** 23](#_Toc27710076)

[**5.2** **Функции семантического анализатора** 23](#_Toc27710077)

[**5.4 Принцип обработки ошибок** 24](#_Toc27710078)

[**5.5 Контрольный пример** 24](#_Toc27710079)

[**6 ПРЕОБРАЗОВАНИЕ ВЫРАЖЕНИЙ** 25](#_Toc27710080)

[**6.1 Выражения, допускаемые языком** 25](#_Toc27710081)

[**6.2 Польская запись** 25](#_Toc27710082)

[**6.3 Программная реализация обработки выражений** 26](#_Toc27710083)

[**6.4 Контрольный пример** 26](#_Toc27710084)

[**7 ГЕНЕРАЦИЯ КОДА** 27](#_Toc27710085)

[**7.1 Структура генератора кода** 27](#_Toc27710086)

[**7.2 Представление типов данных в оперативной памяти** 27](#_Toc27710087)

[**7.3** **Статическая библиотека** 28](#_Toc27710088)

[**7.4** **Алгоритм работы генератора кода** 28](#_Toc27710089)

[**7.3** **Контрольный пример** 28](#_Toc27710090)

[**8 ТЕСТИРОВАНИЕ ТРАНСЛЯТОРА** 29](#_Toc27710091)

[**8.1 Тестирование фазы проверки на допустимость символов** 29](#_Toc27710092)

[**Заключение** 30](#_Toc27710093)

[**Приложение А** 31](#_Toc27710094)

[**Приложение Б** 32](#_Toc27710095)

[**Приложение В** 33](#_Toc27710096)

[**Приложение Г** 35](#_Toc27710097)

[**Приложение Д** 39](#_Toc27710098)

[**Приложение Е** 41](#_Toc27710099)

[**Приложение З** 42](#_Toc27710100)

[**Литература** 45](#_Toc27710101)

# **Введение**

Компилятор – это программа, которая преобразует исходный текст программы, написанный на языке программирования высокого уровня, в программу на машинном языке, «понятную» компьютеру.

Классический компилятор состоит из следующих частей:

– Лексический анализатор;

– Синтаксический анализатор;

– Семантический анализатор;

– Генератор кода.

Все части компилятора взаимодействуют между собой, обрабатывая входной текст и строят из него эквивалентный текст на понятном языке программирования для компьютера.

Цель курсового проекта – разработка всех частей компилятора для собственного языка программирования. В процессе создания компилятора будут проведены многие исследования эффективности тех или иных алгоритмов, будут укрепленные и получены новые знания о структуре языков программирования, и в конечном итоге будет получен требуемый результат, а именно – рабочий проект на собственном языке программирования.

Курсовой проект разбит на 8 глав, в каждой из которых будет подробно описан процесс работы анализаторов и генератора. Также будет представлен новый язык программирования КАО-2019 и код рабочей программы, демонстрирующий все особенности языка. Также будут представлены различные коды языка с ошибочным завершением и подробным объяснением причин этого завершения.

В процессе работы будут проведены все основные этапы разработки программного обеспечения. На этапе анализа будут изучены алгоритмы построения компилятора, на этапе проектирования будет разработана грамматика языка КАО-2019, на этапе реализации будет написан рабочий код, а на этапе тестирования будут выявлены основные недочёты проделанной работы, на этапе сопровождения будет оформлена документация и исправлены все ошибки, найденные ранее.

# **1 СПЕЦИФИКАЦИЯ ЯЗЫКА ПРОГРАММИРОВАНИЯ**

## **1.1 Характеристика языка программирования**

Язык KAО-2019 – это строго типизированный, процедурный, компилируемый язык. Он является транслируемым, не объектно-ориентированным. В KAО-2019 используется три типа данных: целочисленный тип данных int!, строковый тип данных str!, логический тип данных bool!.

## **1.2 Алфавит языка**

Алфавит КАО-2019 представлен из:

* прописных и строчных букв латинского алфавита {A-Za-z};
* арабских цифр {0-9};
* специальных символов {=, +, -, /, \*, %, (, ), {, }, [, ], ?, &, #, <, >};
* ключевых слов {var, func, go, return, out, check, otherwise, int!, str!, bool!, random, strlen, Line}

## **1.3 Применяемые сепараторы**

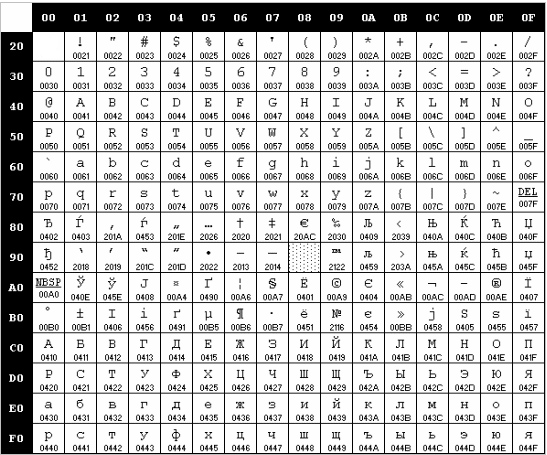
Символы-сепараторы служат для разделения цепочек языка на уровне первичной обработки исходного текста программы.

Таблица 1.1 Применяемые сепараторы

|  |  |  |
| --- | --- | --- |
| Сепаратор | Назначение | Правила применения |
| () | Круглые скобки | Приоритетность операций |
| {} | Фигурные скобки | Программный блок функции |
| & | Запятая | Разделитель параметров функции |
| . | Точка | Разделитель инструкций |
| [] | Квадратные скобки | Параметры функций |

## **1.4 Применяемые кодировки**

В языке КАО-2019 используется кодировка Windows-1251.

Рисунок 1.1 Таблица кодировки Windows-1251

## **1.5 Типы данных**

В языке КАО-2019 предусмотрены типы данных, приведенные в таблице 1.2.

Таблица 1.2 Типы данных

|  |  |
| --- | --- |
| Тип данных | Описание |
| int! | Беззнаковый целочисленный тип данных (4 байта). Допустимый диапазон значений 0 / 4 294 967 295. Автоматически инициализируется нулевым значением.  Поддерживаемые операции:   1. + (бинарный) – оператор сложения; 2. - (бинарный) – оператор вычитания; 3. \* (бинарный) – оператор произведения; 4. / (бинарный) – оператор деления; 5. % – бинарный, остаток от деления 6. = (унарная) – оператор присваивания. |
| str! | Строковый тип данных. Максимальное число символов – 255. Каждый символ занимает 1 байт. Автоматическая инициализация строкой нулевой длины. Признак конца строки "\0".  Поддерживаемые операции:   1. = (унарная) – оператор присваивания. |
| bool! | Логический тип данных. Принимает значение true (1) или false (0). В памяти хранится как целочисленный тип.  Поддерживаемые операции:   1. = (унарная) – оператор присваивания. |

## **1.6 Преобразование типов данных**

Преобразование не поддерживается, все типы данных определены однозначно и не могут быть преобразованы в другие, так как язык KAО-2019 является строго типизируемым.

## **1.7 Идентификаторы**

Идентификаторы в языке КАО-2019 состоят из символов нижнего регистра латинского алфавита. Максимальная длина идентификатора — 10 символов. Зарезервированные идентификаторы предусмотрены для стандартной библиотеки языка.

Примеры правильных идентификаторов: len, boo.

Примеры неправильных идентификаторов: Obj, \_str, 1234, x.1

## **1.8 Литералы**

Литерал – лексема, которая непосредственно представляет некоторое значение. Краткое описание литералов представлено в таблице 1.3.

Таблица 1.3 Описание литералов

|  |  |
| --- | --- |
| Литерал | Пояснение |
| Целочисленные литералы | Целые числа. Допустимый диапазон значений 0/ 4 294 967 295. Имеют десятичное и шестнадцатеричное представления. |
| Строковые литералы | Набор символов, заключённых в двойные кавычки ("), максимальное число которых не может превышать 255. В случае превышения длины литерала работа транслятора прекращается. Могут состоять из всех разрешённых языком символов. |
| Логические литералы | Две логические константы true (1) и false (0). |

## **1.9 Объявление данных**

Переменные объявляются при помощи конструкции:

var <тип данных><идентификатор>.

В языке КАО-2019 требуется обязательное объявление переменной перед её использованием. Все переменные должны находиться внутри программного блока языка. Имеется возможность объявления одинаковых переменных в разных блоках. Каждая переменная получает префикс – название функции, в которой она объявлена.

## **1.10 Инициализация данных**

При объявлении переменной допускается инициализация данных. При этом переменной будет присвоено значение литерала или идентификатора, стоящего справа от знака равенства. Объектами-инициализаторами могут быть идентификаторы, литералы, выражения и вызовы функций.

Для инициализации переменной применяется конструкция:

<идентификатор>=<значение>.

## **1.11 Инструкции языка**

Все возможные инструкции языка программирования КАО-2019 представлены в таблице 1.4.

Таблица 1.4 Инструкции языка

|  |  |
| --- | --- |
| Инструкция языка | Синтаксис |
| Главная функция | go  {  <инструкции языка>  } |
| Вызов функций | <идентификатор функции>[<идентификатор / литерал> & …] |
| Возврат из функции | return [идентификатор / литерал] |
| Объявление функции | func <тип данных> <идентификатор>[<тип данных> <идентификатор> & …]  {  <инструкции языка>  } |
| Объявление переменной | var <тип данных> <идентификатор> |
| Присваивание | <идентификатор>=<значение> |
| Вывод данных | out [<идентификатор / литерал>] |
| Инструкция условия | check [<идентификатор / литерал>[?#<>]<идентификатор / литерал>]  {  …  }  otherwise  {  …  } |

## **1.12 Операции языка**

Наибольшую приоритетность арифметических операций имеют операции умножения, деления, остаток от деления, а сложение, вычитание меньшую. При одинаковом приоритете первой выполнится операция, расположенная левее. Изменить приоритетность можно с помощью круглых скобок.

Операции в языке программирования KAO-2019 применимые к целочисленным типам данных приведены в таблице 1.5.

Таблица 1.5 - Операции языка

|  |  |
| --- | --- |
| Тип оператора | Операторы |
| Арифметические | () – приоритет  **+ –** сложение  - – вычитание  \* – умножение  / – деление  % – остаток от деления  = – присваивание |

## **1.13 Выражения и их вычисление**

В языке KAO-2019 реализованы следующие правила при составлении и вычислении выражений:

– Допускается использовать скобки для составления приоритета;

– Выражение может содержать вызов функции;

– Каждое выражение обязано заканчиваться разделителем инструкций;

– Использование двух подряд идущих операторов не допускается;

– Выражение просматриваются справа налево и выполняются в соответствии с приоритетом;

## **1.14 Конструкции языка**

Ключевые программные конструкции языка программирования КАО-2019 представлены в таблице 1.6.

Таблица 1.6. Программные конструкции языка программирования КАО-2019

|  |  |
| --- | --- |
| Название | Синтаксис |
| Точка входа | go  {  <инструкции языка>  } |

Продолжение таблицы 1.6

|  |  |
| --- | --- |
| Название | Синтаксис |
| Функция | func <тип данных><идентификатор>[<тип данных> <идентификатор> & …]{ } |
| Условный оператор | check [<идентификатор / литерал>[?#<>]<идентификатор / литерал>]  {  …  }  otherwise  {  …  } |

## **1.15 Область видимости идентификаторов**

Область видимости языка программирования KAO-2019 используется по принципу сверху вниз. Все переменные являются видимыми внутри объявленной функции.

## **1.16 Семантические проверки**

Перечень семантических проверок, предусмотренных языком, приведен в таблице 1.7.

Таблица 1.7 - Перечень семантических проверок

|  |  |
| --- | --- |
| Номер | Правило |
| 1 | Все идентификаторы должны быть объявлены до использования. |
| 2 | При объявлении идентификатора должен быть указан тип. |
| 3 | Длина идентификатора должна быть не более 10 символов. |
| 4 | Длина строчного литерала не должна превышать 255 символов без учёта кавычек. |
| 5 | Количество фактических параметров, переданных в функцию должно соответствовать количеству формальных параметров вызываемой функции. |
| 6 | Применение арифметических операций допускается только с целочисленным типом. |
| 7 | Тип формальных параметров функции должен совпадать с типом фактических параметров, переданных в функцию. |
| 8 | Отсутствует точка входа |
| 9 | Обнаружено несколько точек входа |

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

Транслированный код использует две области памяти: сегмент констант (литералы) и сегмент данных (переменные и параметры функций).

## **1.18 Стандартная библиотека и ее состав**

Функции стандартной библиотеки и их описание представлено в таблице 1.8.

Таблица 1.8. Состав стандартной библиотеки

|  |  |  |  |
| --- | --- | --- | --- |
| Имя функции | Возвращаемое значение | Параметры функции | Описание функции |
| Strlen(char\* str) | int | str – строка | Возвращает длину строки |
| Random(int number) | int | number – число | Возвращает рандомное число от 0 до number |
| Line() | void | - | Перевод на новую строку |
| OutInt(int number) | void | number – число | Вывод числа на консоль |
| OutStr(char\* ptr) | Void | ptr – строка | Вывод строки на консоль |

## **1.19 Ввод и вывод данных**

Вывод данных осуществляется программной конструкцией out, в скобках указывается имя идентификатора либо литерала. Вычисление выражений внутри конструкции не предусматривается.

## **1.20 Точка входа**

В языке КАО-2019 точкой входа является главная функция “go”. Выполнение программы начинается с первой инструкции главной функции.

## **1.21 Препроцессор**

В языке KAО-2019 препроцессор не предусмотрен.

## **1.22 Соглашения о вызовах**

При генерации кода используется соглашение stdcall. Особенности stdcall:

1. все параметры функций передаются через стек;

2. память высвобождает вызываемый код;

3. занесение в стек параметров идет справа налево.

## **1.23 Объектный код**

Исходный код языка транслируется в язык ассемблера.

## **1.24 Классификация сообщений транслятора**

Транслятор генерирует сообщения о ошибках пользователю. В соответствии с префиксами будут различаться сообщения, представленные в таблице 1.9.

Таблица 1.9 - Классификация сообщений транслятора

|  |  |
| --- | --- |
| Префикс ошибки | Описание ошибки |
| 0-199 | Сообщение, генерируемое при критической ошибке системы. |
| 200-299 | Сообщение, генерируемое на этапе лексического анализа. |
| 300-599 | Сообщение, генерируемое на этапе синтаксического анализа. |
| 600-900 | Сообщение, генерируемое на этапе семантического анализа. |

## **1.25 Контрольный пример**

Контрольный пример, написанный на языке KAО-2019, представлен в приложении А.

# **2 СТРУКТУРА ТРАНСЛЯТОРА**

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

Исходный код, написанный на языке программирования KAО-2019, является для транслятора входными данными.

Как выходные данные, используется объектный код и протоколы работы транслятора.

Компоненты транслятора приведены на рисунке 2.1.



Рис 2.1 - Структура транслятора KAО-2019

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

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

Наборы функций, проверяющие правила на разных этапах работы транслятора представлены в семантическом анализаторе. Продолжение или остановка работы транслятора всецело зависит от критичности возникающих ошибок.

Генерация кода реализуется посредством чистой интерпретации, без создания промежуточного представления кода. В финале происходит генерация кода, во время исполнения которого формируется объектный код.

## **2.2 Перечень входных параметров транслятора**

В таблице 2.1 представлены входные параметры, которые могут использоваться для представления работы транслятора.

Таблица 2.1 - Входные параметры транслятора

|  |  |  |
| --- | --- | --- |
| Входной параметр | Описание | Значение по умолчанию |
| -in:<имя\_файла> | Входной файл с расширением .txt, в котором содержится исходный код на КАО-2019 | Не предусмотрено |
| -log:<имя\_файла> | Файл для записи результата проверки входного файла на допустимость символов, результатов лексического и семантического анализа, трассировка синтаксического анализа | <имя\_файла>.log |
| -out:<имя\_файла> | Файл для записи результата работы транслятора | <имя\_файла>.asm |

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

Таблица с перечнем протоколов, формируемых транслятором языка КАО-2019 и их назначением представлена в таблице 2.2.

Таблица 2.2 – Протоколы, формируемые транслятором языка КАО-2019

|  |  |
| --- | --- |
| Формируемый протокол | Описание протокола |
| Файл журнала с параметром <log> | Содержит информацию о входных параметрах в приложение и о этапе проверки символов на допустимость, результат работы лексического и семантического анализа. Содержит таблицы лексем и идентификаторов. |
| Выходной файл c параметром <out> | Содержит сгенерированный код на языке Ассемблера. |

# **3 РАЗРАБОТКА ЛЕКСИЧЕСКОГО АНАЛИЗАТОРА**

## **3.1 Структура лексического анализатора**

Структура лексического анализатора приведена на рисунке 3.1.



Рисунок 3.1. Структура лексического анализатора

Лексический анализатор – часть транслятора, выполняющая лексический анализ. На этом этапе распознаётся правильность составления лексем и идентификаторов.

Обязательным входным параметром для лексического анализаторов является файл с исходным кодом программы на языке программирования КАО-2019, имеющий расширение .txt.

Результатом работы лексического анализатора является таблица лексем и идентификаторов, которые заполняются в процессе анализа.

## **3.2 Контроль входных символов**

Исходный код на языке программирования КАО-2019 перед лексическим анализом проверяется на допустимость символов. Файл с исходным кодом считывается посимвольно и каждый символ проверяется на допустимость.

Для проверки символов была создана структура, аналогичная таблице кодировки Windows-1251. Данная структура представлена в приложении Б.

Каждый считанный из файла с исходным кодом символ проверяется на допустимость. Если символ в таблице обозначен как T, то он будет записан в строку, если символ обозначен буквой F, то его позиция в исходном коде записывается в структуру ошибок. В случае обнаружения недопустимого символа будет сгенерирована ошибка и она будет выведена в консоль вместе с позицией в исходном коде.

## **3.3 Удаление избыточных символов**

Избыточными в языке программирования КАО-2019 считаются символы пробела, не влияющие на ход выполнения программы. Удаление избыточных символов осуществляется проверкой исходного кода. Если между словами больше одного пробела, то лишние пробелы будут удаляться.

## **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 3.1.

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

|  |  |  |
| --- | --- | --- |
| Тип цепочки | Цепочка | Лексема |
| Ключевые слова | Var | v |
| int! | t |
| str! | t |
| bool! | t |
| Func | f |
| Return | r |
| Go | g |
| Check | c |
| Otherwise | u |
| Random | ! |
| Strlen | ~ |
| Иное | Идентификатор | i |
| Целочисленный литерал | l |
| Строковый литерал | l |
| Логический литерал | l |
| Сепаторы | . | . |
| & | & |
| ( | ( |
| ) | ) |
| [ | [ |
| ] | ] |
| { | { |
| } | } |

Продолжение таблицы 3.1

|  |  |  |
| --- | --- | --- |
| Тип цепочки | Цепочка | Лексема |
| Операторы | + | s |
| - | s |
| \* | s |
| / | s |
| % | s |
| = | = |
| > | b |
| < | b |
| ? | b |
| # | b |

Пример реализации таблицы лексем представлен в приложении Е.

## **3.5 Основные структуры данных**

В приложении В представлены основные структуры данных на этапе синтаксического анализа.

## **3.6 Принцип обработки ошибок**

При возникновении ошибки работа транслятора прекращается.

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

Перечень сообщений лексического анализатора представлен на рисунке 3.2.

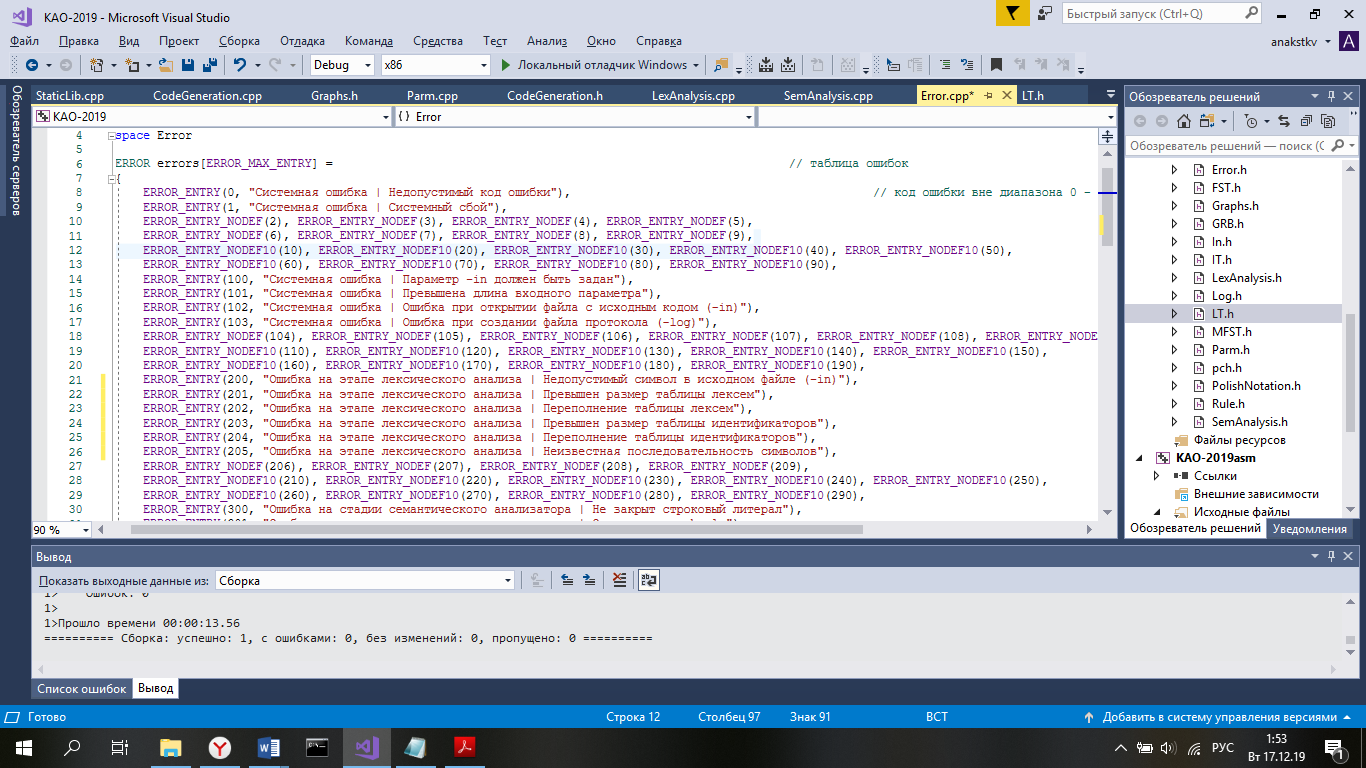


Рисунок 3.2 – Перечень ошибок лексического анализатора

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

Входным параметром лексического анализа является очередь, состоящая из структур, полями которых являются лексема и номер её строки в исходном файле, полученные на этапе проверки исходного кода на допустимость символов.

## **3.9 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.

Пример. Регулярное выражение для ключевого слова go: ‘go’.

Граф конечного автомата для этой лексемы представлен на рисунке 3.3. S0 – начальное состояние, S2 – конечное состояние автомата.

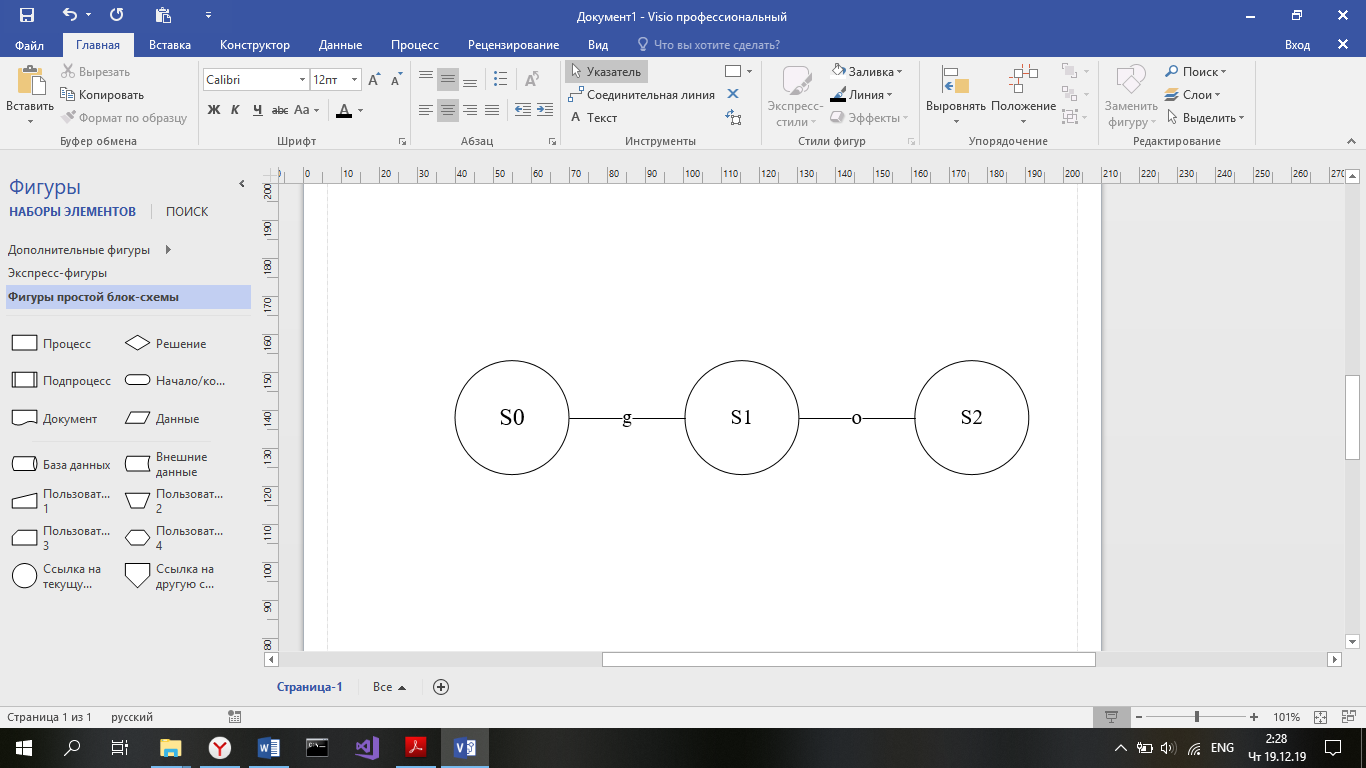


Рисунок 3.3 – Граф переходов для цепочки ‘go’

## **3.10 Контрольный пример**

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

# 

# **4 РАЗРАБОТКА СИНТАКСИЧЕСКОГО АНАЛИЗАТОРА**

## **4.1 Структура синтаксического анализатора**

Синтаксический анализ – это фаза трансляции, выполняемая после лексического анализа и предназначенная для распознавания синтаксических конструкций. 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 KAО-2019 используется контекстно-свободная грамматика , где

T – множество терминальных символов (было описано в разделе 1.2 данной пояснительной записки),

N – множество нетерминальных символов (первый столбец таблицы 4.1),

P – множество правил языка (второй столбец таблицы 4.1),

S – начальный символ грамматики, являющийся нетерминалом.

Эта грамматика имеет нормальную форму Грейбах, т.к. она не леворекурсивная (не содержит леворекурсивных правил) и правила  имеют вид:

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

Грамматика языка KAO-2019 представлена в приложении В в виде заполненной структуры на языке С++. В структуре терминальные символы помечены символами TS и все они являются сепараторами, знаками арифметических операций или строчными буквами. Нетерминальные символы помечены словом NS и являются заглавными буквами латинского алфавита.

## **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 В.

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. |
|  | 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 |

## **Основные структуры данных**

Основные структуры данных синтаксического анализатора представлены в приложении В.

## **Описание алгоритма синтаксического разбора**

Формальное описание алгоритма работы синтаксического анализа в общем виде можно описать следующим образом:

1. В магазин записывается стартовый символ.
2. На основе полученной ранее таблицы лексем формируется входная лента.
3. Запускается автомат.
4. Выбирается цепочка по первому символу, соответствующая нетерминальному символу, записывается в магазин в обратном порядке.
5. Если терминалы в стеке и в ленте совпадают, то данный терминал удаляется из ленты и магазина. Иначе возвращаемся в предыдущее сохраненное состояние и выбираем другое правило нетерминала.
6. Если в правиле встретился нетерминал 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.

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

Перечень сообщений синтаксического анализатора содержится в таблице 4.1

Таблица 4.1 Сообщения синтаксического анализатора

|  |  |
| --- | --- |
| Номер сообщения | Содержание сообщения |
| 600 | Неверная структура программы |
| 601 | Отсутствует список параметров функции при её объявлении |
| 602 | Ошибка в параметрах функции при её объявлении |
| 603 | Возможно отсутствует тело функции |
| 604 | Недопустимое выражение. Ожидаются только литералы и идентификаторы |
| 605 | Неверная конструкция в теле функции |
| 606 | Ошибка в условном выражении |
| 607 | Ошибка в вызове функции |
| 608 | Ошибка в арифметическом выражении |
| 609 | Ошибка в списке параметров при вызове функции |
| 610 | Неверная конструкция в условном выражении |
| 611 | Требуется закрывающаяся фигурная скобка |
| 612 | Требуется открывающаяся фигурная скобка |

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

Входными данными являются таблицы лексем и идентификаторов. Выходными данными синтаксического анализатора являются дерево разбора и трассировка разбора исходного кода.

## **Принцип обработки ошибок**

Изначально, если анализатор разбирает часть исходного кода по какому-либо правилу, то запоминает его. При возникновении ошибки синтаксический анализатор откатиться назад до правила, при помощи которого разбор был успешным, если это возможно. После чего пытается применить последующие правила из грамматики. В случае, если правило невозможно подобрать, то выводится сообщение об ошибке.

## **Контрольный пример**

Пример разбора исходного кода на языке программирования КАО-2019 синтаксическим анализатором представлен в виде дерева разбора, которое находится в приложении Д.

# **5 РАЗРАБОТКА СЕМАНТИЧЕСКОГО АНАЛИЗАТОРА**

## **5.1 Структура семантического анализатора**

Структура семантического анализатора представлена на рисунке 5.1.



Рисунок 5.1. – Структура семантического анализатора

Семантический анализатор – часть транслятора, выполняющая семантический анализ, то есть исходный код проверяется на наличие ошибок, которые невозможно отследить при помощи регулярной и контекстно-свободной грамматики. Входными данными являются таблица лексем и идентификаторов.

Семантический анализатор является отдельной частью транслятора КАО-2019

## **Функции семантического анализатора**

Сообщения семантического анализа ошибки, приведены на рисунке 5.2.

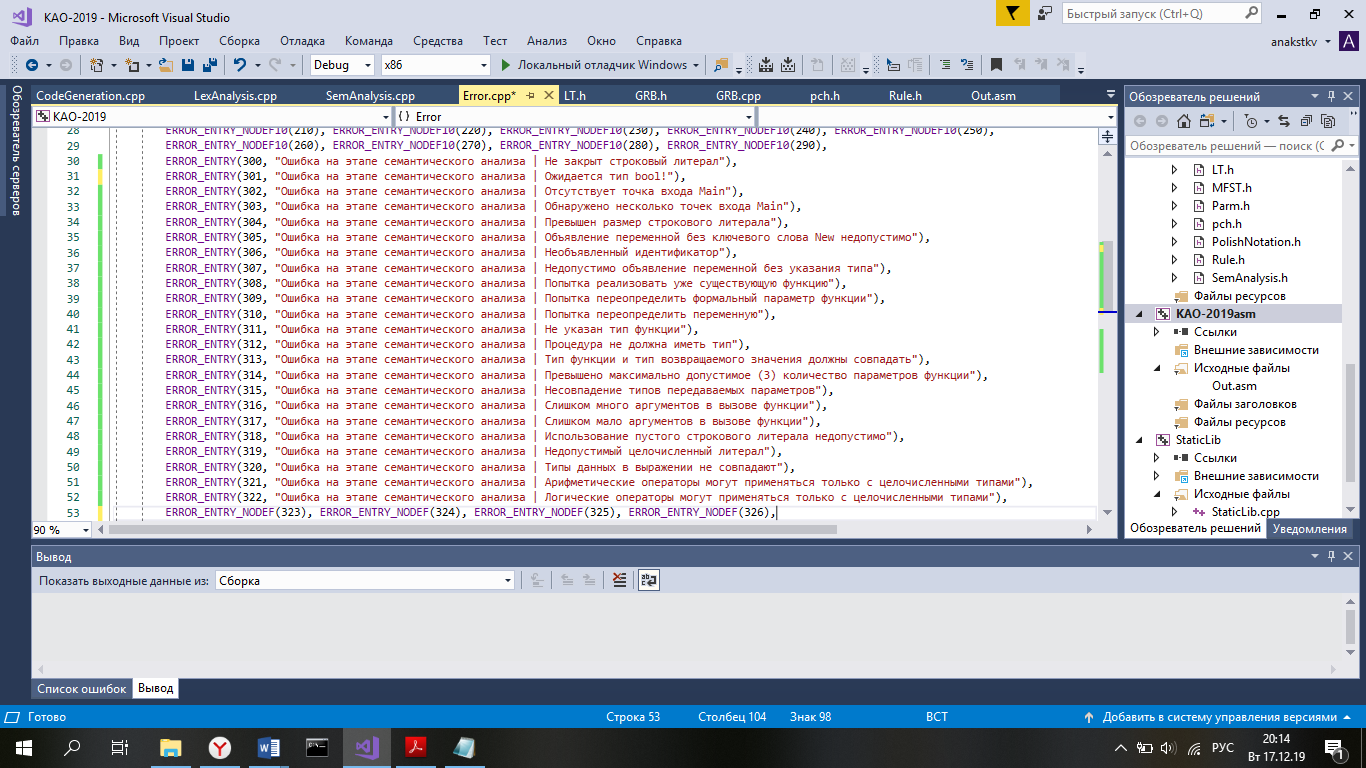


Рисунок 5.2 – Перечень сообщений семантического анализатора

## **5.4 Принцип обработки ошибок**

Принцип обработки ошибок идентичен принципу обработки ошибок на этапе лексического анализа (раздел 3.6).

## **5.5 Контрольный пример**

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

# **6 ПРЕОБРАЗОВАНИЕ ВЫРАЖЕНИЙ**

## **6.1 Выражения, допускаемые языком**

В языке KAO-2019 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 – Приоритет операций в языке KAO-2019

|  |  |
| --- | --- |
| Приоритет | Операция |
| 0 | ( |
| 0 | ) |
| 0 | [ |
| 0 | ] |
| 1 | & |
| 2 | + |
| 2 | - |
| 4 | \* |
| 4 | / |
| 4 | % |

## **6.2 Польская запись**

Выражения в языке КАО-2019 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 и логических выражений, в которой операнды расположены перед знаками операций. 

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;

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

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\*s | - |
| i) | q2\*s( | - |
| ) | q2\*s(i | - |
|  | q2\*s(i)- |  |

## **6.3 Программная реализация обработки выражений**

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

## **6.4 Контрольный пример**

Пример преобразования выражения к польской записи представлен в таблице 6.4. Преобразование выражений в формат польской записи необходимо для построения более простых алгоритмов их вычисления.

В приложении Г приведены изменённые таблицы лексем и идентификаторов, отображающие результаты преобразования выражений в польский формат.

# **7 ГЕНЕРАЦИЯ КОДА**

## **7.1 Структура генератора кода**

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



Рисунок 7.1 Структура генератора кода

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

Элементы таблицы идентификаторов расположены в разных сегментах языка ассемблера – .data и .const. Идентификаторы языка КАО-2019размещены в сегменте данных(.data). Литералы – в сегменте констант (.const). Соответствия между типами данных идентификаторов на языке КАО-2019и на языке ассемблера приведены в таблице 7.1.

Таблица 7.1 Соответствия типов идентификаторов языка КАО-2019 и языка Ассемблера

|  |  |  |
| --- | --- | --- |
| Тип идентификатора на языке КАО-2019 | Тип идентификатора на языке ассемблера | Пояснение |
| int! | SDWORD | Хранит целочисленный тип данных. |
| str! | BYTE для литералов, DWORD для переменных | Литерал хранит последовательность байтов.  Переменные хранят указатель на начало строки литерала. |
| bool! | SDWORD | Хранит логический тип данных. |

## **Статическая библиотека**

Статическая библиотека написана на языке программирования C++. Подробное описание функций приведено в пункте 1.18. Подключается библиотека с помощью команды includelib <имя.lib>. В процессе компоновки статическая библиотека подключается к выходному файлу.

## **Алгоритм работы генератора кода**

Преобразование происходит по принципу встречи определённой лексемы и знания местоположения данного кода в языке assembler.

На рисунке 7.2. приведён пример преобразования операций сложения и вычитания в язык assembler

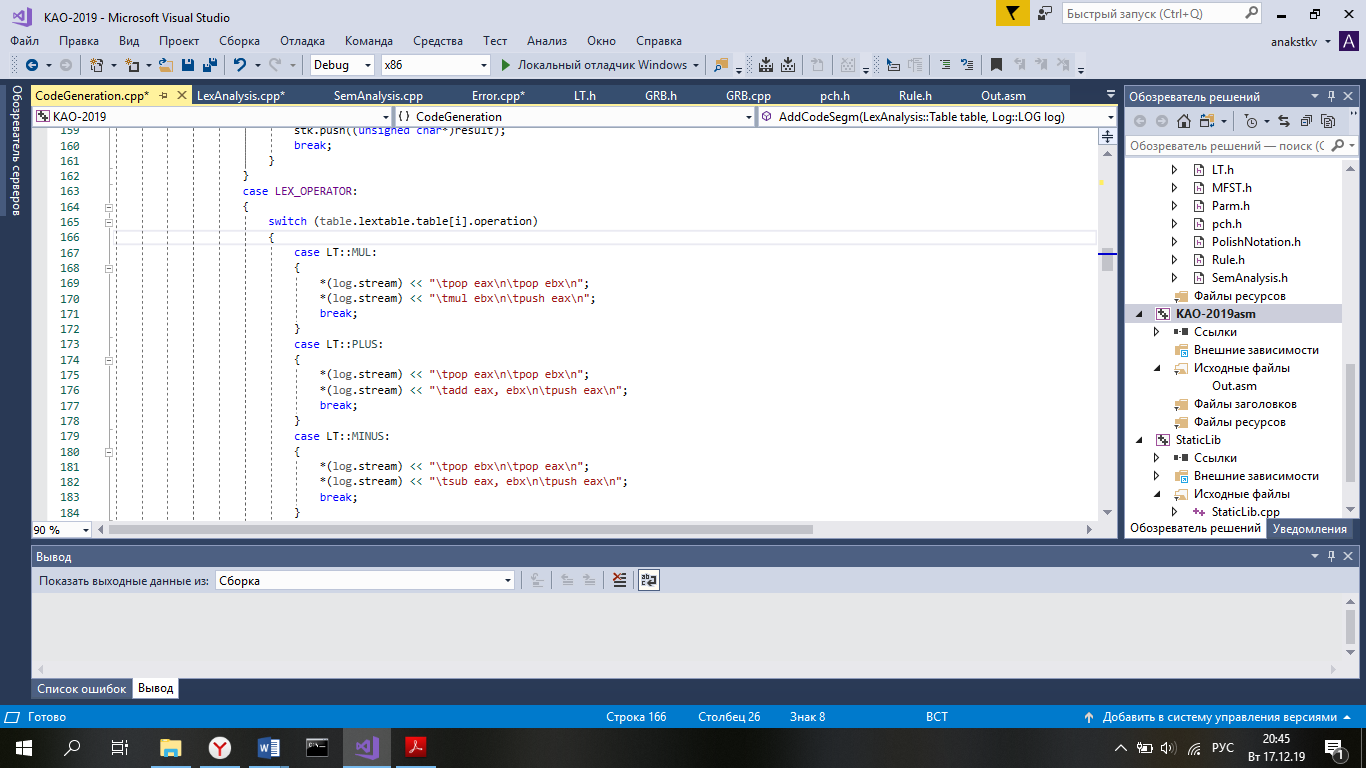


Рисунок 7.2. Преобразование операций сложения и вычитания в язык assembler

## **Контрольный пример**

Контрольный пример генерации кода представлен в приложении З.

# **8 ТЕСТИРОВАНИЕ ТРАНСЛЯТОРА**

## **8.1 Тестирование фазы проверки на допустимость символов**

В результате работы транслятора могут возникнуть ошибки, которые могут повлиять на транслятор и выполняемую программу. Смоделированные ошибки приведены в таблице 8.1.

Таблица 8.1. Ошибки и пути их разрешения

|  |  |  |
| --- | --- | --- |
| Код | Тип ошибки | Выходной результат |
| var int! result12. | Лексическая | Error205 Ошибка на этапе лексического анализа | Неизвестная последовательность символов | Строка: 3 | Позиция в строке: 8 |
| result = one + two) \* two. | Синтаксическая | Error608 Ошибка на этапе синтаксического анализа | Ошибка в арифметическом выражении |
| var int! f = math [2 & ] | Семантическая | Error317 Ошибка на этапе семантического анализа | Слишком мало аргументов в вызове функций |

# **Заключение**

В ходе выполнения курсовой работы был разработан транслятор для языка программирования KAO-2019. Таким образом, были выполнены основные задачи данной курсовой работы:

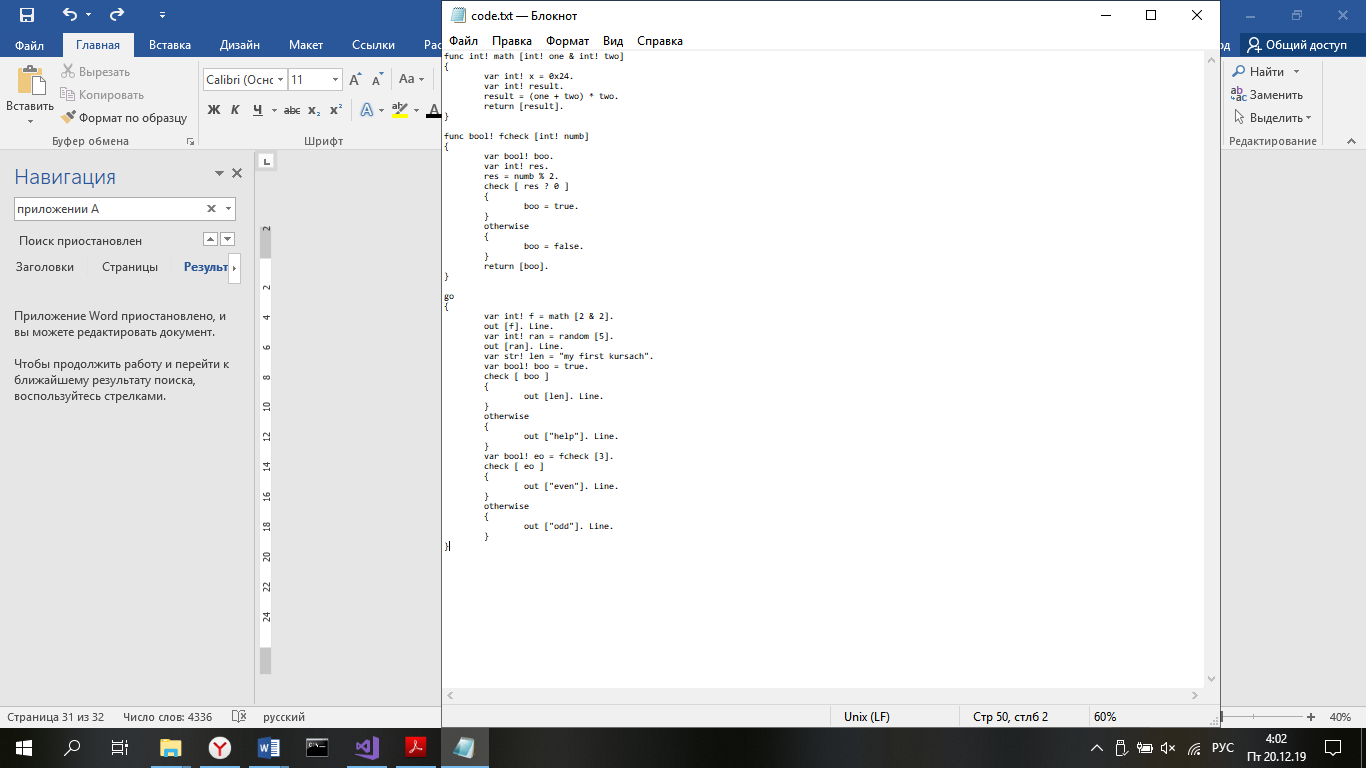
1. Сформулирована спецификация языка KAO-2019;
2. Разработаны конечные автоматы и алгоритмы для реализация лексического анализатора;
3. Разработана контекстно-свободная, приведённая к ослабленной нормальной форме Грейбах, грамматика для описания синтаксически верных конструкций языка;
4. Разработан семантический анализатор, осуществляющий проверку используемых инструкций;
5. Разработан транслятор с языка программирования KAO-2019 на язык низкого уровня Assembler;
6. Проведено тестирование всех вышеперечисленных компонентов.

Окончательная версия языка KAO-2019 включает:

1. 3 типа данных
2. поддержка операции вывода
3. возможность вызова функций стандартной библиотеки
4. наличие 5 арифметических операторов для вычисления выражений
5. Условный оператор
6. Структурированная система для обработки ошибок пользователя.
7. Поддержка функций

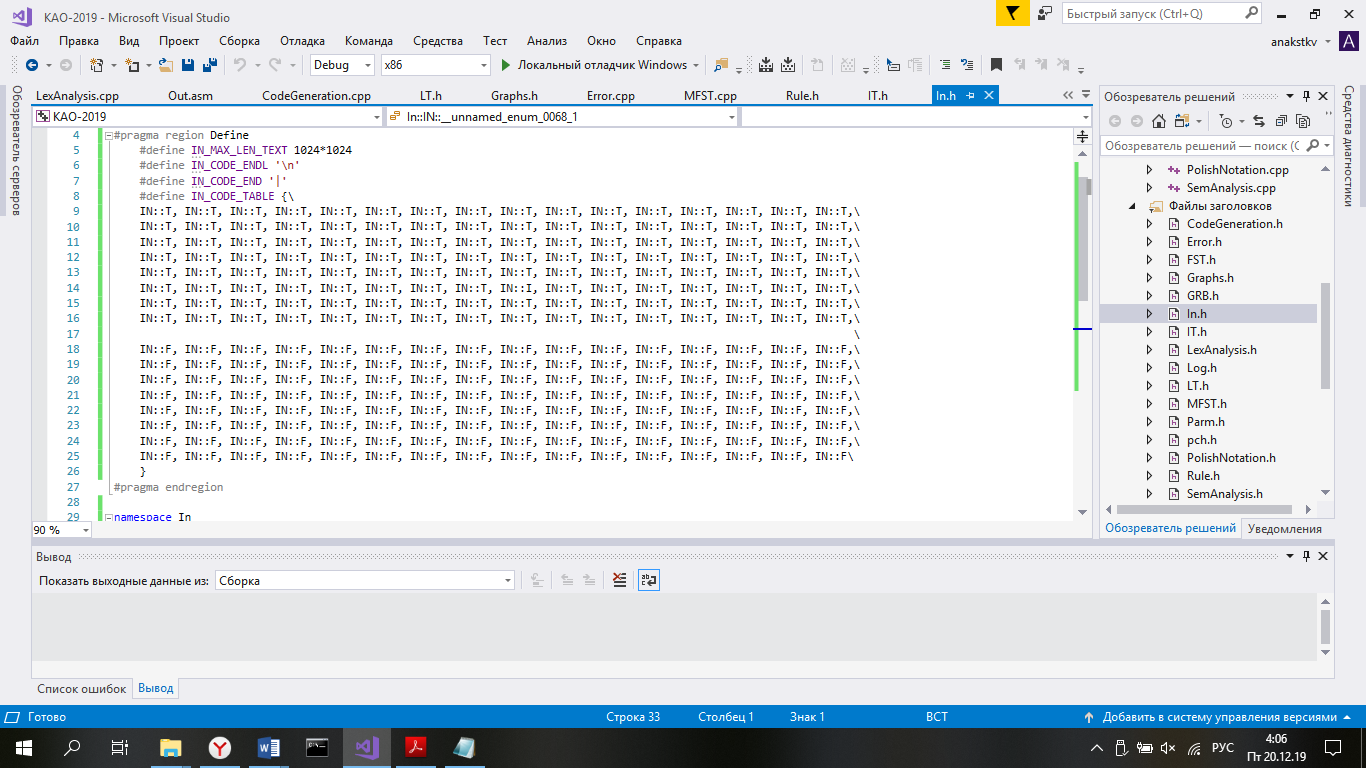
# **Приложение А**

**Контрольный пример**



# **Приложение Б**

**Таблица входных символов**



# **Приложение В**

**Структура синтаксического анализатора**

struct MfstState

{

short lenta\_position;

short nrulechain;

short nrule;

MFSTSTSTACK st;

MfstState();

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

MfstState(

short pposition,

MFSTSTSTACK pst,

short pnrule,

short pnrulechain

);

};

struct Mfst

{

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;

int FST\_TRACE\_n = 0;

short lenta\_position;

short nrule;

short nrulechain;

short lenta\_size;

GRB::Greibach grebach;

LT::LexTable lexTable;

MFSTSTSTACK st;

std::stack<MfstState> storestate;

Mfst();

Mfst(LT::LexTable& lexTable, GRB::Greibach pgrebach);

char\* getCSt(char\* buf);

char\* getCLenta(char\* buf, short pos, short n = 25);

char\* getDiagnosis(short n, char\* buf);

bool savestate(Log::LOG log);

bool restate(Log::LOG log);

bool push\_chain(GRB::Rule::Chain chain);

RC\_STEP step(Log::LOG log);

bool start(Log::LOG log);

bool savediagnosis(RC\_STEP pprc\_step);

void printrules(Log::LOG log);

struct Deducation

{

short size;

short\* nrules;

short\* nrulechains;

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

} deducation;

bool savededucation();

};

# **Приложение Г**

**Код программы, приводящей выражения к обратной польской записи**

namespace Polish

{

bool isPolish(int i, LexAnalysis::Table& lex)

{

std::stack<LT::Entry> stack;

std::queue<LT::Entry> queue;

LT::Entry temp;

temp.idxTI = LT\_TI\_NULLIDX;

temp.lexema = '#';

temp.line = lex.lextable.table[i].line;

LT::Entry func;

func.idxTI = LT\_TI\_NULLIDX;

func.lexema = '@';

func.line = lex.lextable.table[i].line;

int countLex = 0, countParm = 1, posLex = i;

char\* buf = new char[1];

bool findComma = false;

for (i; lex.lextable.table[i].lexema != LEX\_POINT; i++, countLex++)

{

switch (lex.lextable.table[i].lexema)

{

case LEX\_ID:

{

queue.push(lex.lextable.table[i]);

continue;

}

case LEX\_RANDOM:

{

queue.push(lex.lextable.table[i]);

continue;

}

case LEX\_STRLEN:

{

queue.push(lex.lextable.table[i]);

continue;

}

case LEX\_LITERAL:

{

queue.push(lex.lextable.table[i]);

continue;

}

case LEX\_OPERATOR:

{

while (!stack.empty() && lex.lextable.table[i].priority <= stack.top().priority)

{

queue.push(stack.top());

stack.pop();

}

stack.push(lex.lextable.table[i]);

continue;

}

case LEX\_AND:

{

findComma = true;

continue;

}

case LEX\_LEFTBRACKET:

{

stack.push(lex.lextable.table[i]);

continue;

}

case LEX\_RIGHTBRACKET:

{

while (stack.top().lexema != LEX\_LEFTBRACKET)

{

queue.push(stack.top());

stack.pop();

if (stack.empty())

return false;

}

stack.pop();

continue;

}

case LEX\_LEFTTHESIS:

{

stack.push(lex.lextable.table[i]);

continue;

}

case LEX\_RIGHTTHESIS:

{

while (stack.top().lexema != LEX\_LEFTTHESIS)

{

queue.push(stack.top());

stack.pop();

if (stack.empty())

return false;

}

if (findComma) countParm++;

lex.lextable.table[i] = func;

queue.push(lex.lextable.table[i]);

\_itoa\_s(countParm, buf, 2, 10);

stack.top().lexema = buf[0];

stack.top().idxTI = func.idxTI;

stack.top().line = func.line;

queue.push(stack.top());

stack.pop();

continue;

}

}

}

while (!stack.empty())

{

if (stack.top().lexema == LEX\_LEFTTHESIS || stack.top().lexema == LEX\_RIGHTTHESIS || stack.top().lexema == LEX\_LEFTBRACKET || stack.top().lexema == LEX\_RIGHTBRACKET)

return false;

queue.push(stack.top());

stack.pop();

}

while (countLex)

{

if (!queue.empty())

{

lex.lextable.table[posLex++] = queue.front();

queue.pop();

}

else

{

lex.lextable.table[posLex++] = temp;

}

countLex--;

}

for (int i = 0; i < posLex; i++)

{

if (lex.lextable.table[i].lexema == LEX\_LITERAL)

lex.idtable.table[lex.lextable.table[i].idxTI].idxfirstLE = i;

}

}

bool GoPolish(LexAnalysis::Table& lex)

{

bool rc = false;

bool equ = false;

for (int i = 0; i < lex.lextable.size; i++)

{

if (lex.lextable.table[i].lexema == LEX\_EQUAL || (lex.lextable.table[i].lexema == LEX\_ID && lex.lextable.table[i - 1].lexema != LEX\_EQUAL &&

lex.idtable.table[lex.lextable.table[i].idxTI].idtype == IT::F && lex.lextable.table[i - 2].lexema != LEX\_FUNCTION))

{

rc = isPolish(i + 1, lex);

if (!rc) return rc;

}

}

return rc;

}

}

fti[ti&ti]

2 {

3 vti=l.

4 vti.

5 i=iisis##.

6 r[i].

7 }

9 fti[ti]

10 {

11 vti.

12 vti.

13 i=ils.

14 c[ibl]

15 {

16 i=l.

17 }

18 u

19 {

20 i=l.

21 }

22 r[i].

23 }

25 g

26 {

27 vti=ill@2#.

28 o[i].n.

29 vti=!l@1.

30 o[i].n.

31 vti=l.

32 vti=l.

33 c[i]

34 {

35 o[i].n.

36 }

37 u

38 {

39 o[l].n.

40 }

41 vti=il@1.

42 c[i]

43 {

44 o[l].n.

45 }

46 u

47 {

48 o[l].n.

49 }

50 }

# **Приложение Д**

**Дерево разбора**

0000 | S->ftiFBS

0003 | F->[P]

0004 | P->ti&P

0007 | P->ti

0010 | B->{Nr[I].}

0011 | N->vti=E.N

0015 | E->l

0017 | N->vti.N

0021 | N->i=E.

0023 | E->(E)M

0024 | E->iM

0025 | M->sE

0026 | E->i

0028 | M->sE

0029 | E->i

0033 | I->i

0037 | S->ftiFBS

0040 | F->[P]

0041 | P->ti

0044 | B->{Nr[I].}

0045 | N->vti.N

0049 | N->vti.N

0053 | N->i=E.N

0055 | E->iM

0056 | M->sE

0057 | E->l

0059 | N->c[R]{X}u{X}

0061 | R->ibl

0066 | X->i=E.

0068 | E->l

0073 | X->i=E.

0075 | E->l

0080 | I->i

0084 | S->g{N}

0086 | N->vti=E.N

0090 | E->iK

0091 | K->[W]

0092 | W->l&W

0094 | W->l

0097 | N->o[I].N

0099 | I->i

0102 | N->n.N

0104 | N->vti=E.N

0108 | E->!K

0109 | K->[W]

0110 | W->l

0113 | N->o[I].N

0115 | I->i

0118 | N->n.N

0120 | N->vti=E.N

0124 | E->l

0126 | N->vti=E.N

0130 | E->l

0132 | N->c[R]{X}u{X}N

0134 | R->i

0137 | X->o[I].N

0139 | I->i

0142 | N->n.

0147 | X->o[I].N

0149 | I->l

0152 | N->n.

0155 | N->vti=E.N

0159 | E->iK

0160 | K->[W]

0161 | W->l

0164 | N->c[R]{X}u{X}

0166 | R->i

0169 | X->o[I].N

0171 | I->l

0174 | N->n.

0179 | X->o[I].N

0181 | I->l

0184 | N->n**.**

# **Приложение Е**

**Таблица лексем**

1 fti[ti&ti]

2 {

3 vti=l.

4 vti.

5 i=(isi)si.

6 r[i].

7 }

9 fti[ti]

10 {

11 vti.

12 vti.

13 i=isl.

14 c[ibl]

15 {

16 i=l.

17 }

18 u

19 {

20 i=l.

21 }

22 r[i].

23 }

25 g

26 {

27 vti=i[l&l].

28 o[i].n.

29 vti=![l].

30 o[i].n.

31 vti=l.

32 vti=l.

33 c[i]

34 {

35 o[i].n.

36 }

37 u

38 {

39 o[l].n.

40 }

41 vti=i[l].

42 c[i]

43 {

44 o[l].n.

45 }

46 u

47 {

48 o[l].n.

49 }

50 }

# **Приложение З**

**Генерация кода**

.586

.model flat, stdcall

includelib libucrt.lib

includelib kernel32.lib

includelib ../Debug/StaticLib.lib

ExitProcess PROTO :DWORD

EXTRN Random: proc

EXTRN Strlen: proc

EXTRN Line: proc

EXTRN OutInt: proc

EXTRN OutStr: proc

.stack 4096

.const

divisionByZero db 'ERROR: DIVIDE BY ZERO', 0

belowZeroNum db 'ERROR: NUM IS BELOVER THAN ZERO', 0

L1 SDWORD 0

L2 SDWORD 2

L3 SDWORD 1

L4 SDWORD 0

L5 SDWORD 5

L6 BYTE "my first kursach", 0

L7 BYTE "help", 0

L8 SDWORD 3

L9 BYTE "even", 0

L10 BYTE "odd", 0

.data

mathx SDWORD 0

mathresult SDWORD 0

fcheckboo SDWORD 0

fcheckres SDWORD 0

gof SDWORD 0

goran SDWORD 0

golen DWORD ?

goboo SDWORD 0

goeo SDWORD 0

.code

math PROC mathone : DWORD, mathtwo : DWORD

push L1

pop mathx

push mathone

push mathtwo

pop eax

pop ebx

add eax, ebx

push eax

push mathtwo

pop eax

pop ebx

mul ebx

push eax

pop mathresult

push mathresult

jmp local0

local0:

pop eax

ret

math ENDP

fcheck PROC fchecknumb : DWORD

push fchecknumb

push L2

pop ebx

mov edx, 0

pop eax

idiv ebx

push edx

mov eax, edx

pop fcheckres

mov eax, fcheckres

cmp eax, L1

jz m0

jnz m1

je m1

m0:

push L3

pop fcheckboo

jmp e0

m1:

push L4

pop fcheckboo

e0:

push fcheckboo

jmp local1

local1:

pop eax

ret

fcheck ENDP

main PROC

push math

push L2

push L2

pop edx

pop edx

push L2

push L2

call math

push eax

pop gof

push gof

call OutInt

call Line

push L5

pop edx

push L5

call Random

push eax

pop goran

push goran

call OutInt

call Line

push offset L6

pop golen

push L3

pop goboo

mov eax, goboo

cmp eax, 1

jz m2

jnz m3

je m3

m2:

push golen

call OutStr

call Line

jmp e1

m3:

push offset L7

call OutStr

call Line

e1:

push fcheck

push L8

pop edx

push L8

call fcheck

push eax

pop goeo

mov eax, goeo

cmp eax, 1

jz m4

jnz m5

je m5

m4:

push offset L9

call OutStr

call Line

jmp e2

m5:

push offset L10

call OutStr

call Line

e2:

finish:

push offset divisionByZero

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. Смелов, В.В. Курс лекций по предмету “Языки программирования” – 2016

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