

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 GAS-2023»

Выполнил студент Городилина Анастасия Сергеевна
(Ф.И.О.)
Руководитель проекта преп.-стажер Север Александра Сергеевна
(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.)
Консультанты преп.-стажер Север Александра Сергеевна
(учен. степень, звание, должность, подпись, Ф.И.О.)
Нормоконтролер преп.-стажер Север Александра Сергеевна
(учен. степень, звание, должность, подпись, Ф.И.О.)
Курсовой проект защищен с оценкой _____

Минск 2023

Содержание

Введение.....	6
1 Спецификация языка программирования.....	7
1.1 Характеристика языка программирования.....	7
1.2 Определение алфавита языка программирования.....	7
1.3 Применяемые сепараторы.....	7
1.4 Применяемые кодировки	7
1.5 Типы данных	7
1.6 Преобразование типов данных	8
1.7 Идентификаторы	8
1.8 Литералы	9
1.9 Объявление данных.....	9
1.10 Инициализация данных.....	10
1.11 Инструкции языка	10
1.12 Операции языка	11
1.13 Выражения и их вычисления	11
1.14 Программные конструкции языка	11
1.15 Область видимости идентификаторов	12
1.16 Семантические проверки	12
1.17 Распределение оперативной памяти на этапе выполнения	12
1.18 Стандартная библиотека и её состав	13
1.19 Ввод и вывод данных	13
1.20 Точка входа.....	14
1.21 Препроцессор	14
1.22 Соглашения о вызовах	14
1.23 Объектный код	14
1.24 Классификация сообщений транслятора	14
1.25 Контрольный пример	14
2 Структура транслятора	15
2.1 Компоненты транслятора, их назначение и принципы взаимодействия	15
2.2 Перечень входных параметров транслятора	16
2.3 Перечень протоколов, формируемых транслятором и их содержимое	16
3 Разработка лексического анализатора.....	18
3.1 Структура лексического анализатора.....	18
3.2. Входные и выходные данные лексического анализатора	18
3.3 Параметры лексического анализатора.....	19
3.4 Алгоритм лексического анализа	19

3.5. Контроль входных символов	19
3.6 Удаление избыточных символов	20
3.7 Перечень ключевых слов	20
3.8 Основные структуры данных.....	22
3.9 Структура и перечень сообщений лексического анализа	23
3.10 Принцип обработки ошибок	23
3.11 Контрольный пример	23
4 Разработка синтаксического анализатора	24
4.1 Структура синтаксического анализатора	24
4.2 Контекстно-свободная грамматика, описывающая синтаксис	24
4.3 Построение конечного магазинного автомата	26
4.4 Основные структуры данных.....	27
4.5 Описание алгоритма синтаксического разбора	27
4.6 Параметры синтаксического анализатора	27
4.7 Структура и перечень сообщений синтаксического анализатора.....	27
4.8 Принцип обработки ошибок	28
4.9 Контрольный пример	28
5 Разработка семантического анализатора.....	29
5.1 Структура семантического анализатора.....	29
5.2 Функции семантического анализатора	29
5.3 Структура и перечень семантических ошибок.....	29
5.4 Принцип обработки ошибок	30
5.5 Контрольный пример	30
6 Вычисление выражений.....	32
6.1 Выражения, допускаемые языком	32
6.2 Польская запись и принцип ее построения	32
6.3 Программная реализация обработки выражений	33
6.4 Контрольный пример	33
7 Генерация кода.....	34
7.1 Структура генератора кода	34
7.2 Представление типов данных в оперативной памяти	34
7.3 Статическая библиотека.....	34
7.4 Особенности алгоритма генерации кода	35
7.5 Входные параметры генератора кода	36
7.6 Контрольный пример	36
8 Тестирование транслятора.....	37
8.1 Общие положения	37

8.2 Результаты тестирования	37
Заключение	39
Список использованных источников	40
Приложение А.....	41
Приложение Б	42
Приложение В.....	47
Приложение Г	50
Приложение Д.....	54
Приложение Е	57

Введение

В данном курсовом проекте поставлена задача разработки собственного языка программирования и транслятора для него. Название языка – GAS-2023. Написание транслятора будет осуществляться на языке C++, при этом код на языке GAS-2023 будет транслироваться в язык ассемблера. Язык ассемблера – это машинно-ориентированный язык, представляющий формат записи машинных команд, которые понятны для восприятия человеком.

Задание на курсовой проект можно разделить на следующие задачи:

1. Разработка спецификации языка программирования.
2. Разработка структуры транслятора.
3. Разработка лексического анализатора.
4. Разработка синтаксического анализатора.
5. Разработка семантического анализатора.
6. Обработка выражений с помощью польской инверсии.
7. Генерация кода на язык ассемблера.
8. Тестирование транслятора.

1 Спецификация языка программирования

1.1 Характеристика языка программирования

Язык программирования GAS-2023 – это процедурный язык высокого уровня, который транслируется в язык ассемблера. Он строго типизируемый.

1.2 Определение алфавита языка программирования

Символы, используемые на этапе выполнения: [a...z], [A...Z], [0...9], символы пробела и перевода строки, спецсимволы: [] () , ; # + - > < = & !.

1.3 Применяемые сепараторы

Символы-сепараторы необходимы для разделения операция языка. Сепараторы, используемые в языке программирования GAS-2023, приведены в таблице 1.1.

Таблица 1.1 Символы-сепараторы

Символ(ы)	Назначение
‘пробел’	Разделитель цепочек. Допускается везде кроме названий идентификаторов и ключевых слов
()	Параметры операций и функций
{ }	Программный блок инструкций
,	Разделитель параметров функций
+ -	Арифметические операции
< > & !	Операции сравнения
;	Разделитель программных конструкций
=	Оператор присваивания

Важно отметить, что эти символы имеют специальное назначение и использование в языке программирования GAS-2023. Каждый символ выполняет определенную функцию при разделении операций и структур языка.

1.4 Применяемые кодировки

Для написания программ язык GAS-2023 использует кодировку Windows-1251, содержащую английский алфавит, русский алфавит, а также некоторые специальные символы, такие как [] () , ; + - / * > < = & !{ }.

1.5 Типы данных

В языке программирования GAS-2023 используются три типа данных, которые описываются в таблице 1.2. Пользовательские типы данных не поддерживаются.

Таблица 1.2 – Типы данных

Тип данных	Описание типа данных
Целочисленный тип данных uint(1 байт)	Фундаментальный тип данных, используемый для объявления целочисленных данных. Этот тип данных занимает 4 байта. Без явно указанной инициализации переменной, присваивается нулевое значение. Представляет только положительное целое число. Максимальное значение: 4294967295. Минимальное значение: 0.
Строковый тип данных str	Фундаментальный тип данных, используемый для объявления строк. Без явно указанной инициализации переменной, присваивается нулевое значение (пустая строка). Используется для работы с символами, каждый из которых занимает 1 байт. Максимальное количество символов – 255.
Логический тип данных bool	Фундаментальный тип данных. Без явно указанной инициализации переменной, присваивается значение false. Принимает два значения true или false.

Выбор подходящего типа данных важен для эффективного использования памяти и правильной работы программы. При разработке приложений на GAS-2023 необходимо учитывать диапазон значений и требования к точности данных, чтобы выбрать подходящий тип данных для каждой переменной.

1.6 Преобразование типов данных

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

1.7 Идентификаторы

Общее количество идентификаторов ограничено максимальным размером таблицы идентификаторов (4096). Идентификаторы могут содержать только символы нижнего регистра. Максимальная длина идентификатора равна 10 символам. Идентификаторы, объявленные внутри функционального блока, получают префикс, идентичный имени функции, внутри которой они объявлены. Данные правила действуют для всех идентификаторов. Зарезервированные идентификаторы не предусмотрены. Идентификаторы не должны совпадать с ключевыми словами. Типы идентификаторов: имя переменной, имя функции, параметр функции. Имя идентификатора составляется по следующим образом:

- Состоит из символов латинского алфавита нижнего регистра;
- Максимальная длина идентификатора равна 10 и не должна превышать это значение. При превышении максимального значения выбрасывается ошибка.

1.8 Литералы

В языке программирования GAS-2023 существует 3 типа литералов: целые, строковые и логические. Все литералы являются rvalue. Целочисленные литералы могут быть представлены в виде десятичного и восьмеричного представления, а строковые и логические литералы – произвольно. Их краткое описание представлено в таблице 1.3.

Таблица 1.3 – Литералы

Литералы	Пояснение
Целочисленные литералы в десятичном представлении	Последовательность цифр 0...9 без знака минус
Целочисленные литералы в двоичном представлении	Последовательность цифр 0, 1 с предшествующим двумя символом “0b”
Целочисленные литералы в двоичном представлении	Последовательность цифр 0...9 и последовательность букв A...F, a...f с предшествующим двумя символом “16x”
Строковые литералы	Набор символов алфавита языка, заключенных в одинарные кавычки
Логические литералы	true или false

При использовании литералов в программе на GAS-2023 важно соблюдать синтаксические правила и правильно указывать префиксы и заключения в кавычки в соответствии с требованиями языка. Это позволит корректно интерпретировать и использовать значения литералов в программе.

1.9 Объявление данных

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

Пример объявления числового типа с инициализацией:

```
create uint num10 = 100
create uint num16 = 16x15
create uint num2 = 0b1010
```

Пример объявления переменной строкового типа с инициализацией:

```
create str strhello = 'Привет'
create str string = 'good cat'
```

Пример объявления переменной логического типа с инициализацией:

```
create bool booltrue = true
create bool boolfalse = false
```

Для объявления функций используется ключевое слово `method`, перед которым указывается тип функции. Далее обязателен список параметров и тело функции. Все функции должны возвращать значение.

1.10 Инициализация данных

При объявлении переменной допускается инициализация данных. При этом переменной будет присвоено значение литерала или идентификатора, стоящего справа от знака равенства. Способы инициализации переменных языка программирования GAS-2023 представлены в таблице 1.4.

Таблица 1.4 – Способы инициализации переменных

Вид инициализации	Примечание
create <тип данных> <идентификатор> = <значение>;	Инициализация переменной с присваиванием значения.
create <тип данных> <идентификатор>;	Автоматическая инициализация переменной. uint – инициализируется нулем, str – пустой строкой, bool – false.

Инициализация переменных позволяет задать начальные значения и установить изначальное состояние переменных в программе. Это помогает избежать неопределенного поведения и обеспечивает более надежное выполнение программы.

1.11 Инструкции языка

Все возможные инструкции языка программирования GAS-2023 представлены в общем виде в таблице 1.5.

Таблица 1.5 – Инструкции языка программирования GAS-2023

Инструкция	Запись
Объявление переменной	create <тип данных> <идентификатор>; create <тип данных> <идентификатор> = <выражение>;
Присваивание	<идентификатор> = <выражение>;
Объявление функции	<тип данных> method <идентификатор> ([<тип данных> <идентификатор>][, <тип данных> <идентификатор>]*) {...}
Блок инструкций	{ ... }
Возврат из подпрограммы	push <литерал> <идентификатор>;
Вывод данных	write <идентификатор> <литерал>;
Вывод данных с переводом на новую строку	writeline <идентификатор> <литерал>;
Условный оператор	if (<условие>) { ... }

Окончание таблицы 1.5

Инструкция	Запись
Условный оператор с блоком else	<pre>if (<условие>) { ... } else { ... }</pre>

Из представленной таблицы видно, что инструкции языка программирования GAS-2023 включают объявление переменных, присваивание значений, объявление функций, операторы вывода данных и условные операторы. Эти инструкции обеспечивают основные возможности языка для создания структурированных программ.

1.12 Операции языка

Приоритетность операции умножения выше приоритета операций вычитания и сложения. Для установки наивысшего приоритета используются круглые скобки. К строкам нельзя применять арифметические операции. Язык программирования GAS-2023 может выполнять арифметические и сдвиговые операции, представленные в таблице 1.6.

Таблица 1.6 – Операции и их приоритеты

Операция	Приоритет операции
()	0
+ -	1
left right	2

Операции умножения и деления имеют более высокий приоритет, чем сложение и вычитание, и для установки наивысшего приоритета можно использовать круглые скобки.

1.13 Выражения и их вычисления

Круглые скобки в выражении используются для изменения приоритетов операций. Не допускается запись двух подряд арифметических операций. Также круглые скобки могут использоваться для передачи параметров функций. Фигурные скобки содержат блоки кода функций и циклов.

1.14 Программные конструкции языка

Ключевые программные конструкции языка программирования GAS-2023 представлены в таблице 1.7

Таблица 1.7 – Программные конструкции

Главная функция (точка входа в приложение)	main {...}
Функция	<тип данных> method <идентификатор> ([<тип данных> <идентификатор>][, <тип данных> <идентификатор>]*) { ... push <литерал> <идентификатор>; }
Условный оператор	if [<условие>] { ... } else { ... }

Эти программные конструкции предоставляют основу для структурирования программ и выполнения различных операций в языке GAS-2023.

1.15 Область видимости идентификаторов

В языке программирования GAS-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. Объявление глобальных переменных и пользовательских областей не предусмотрено.

1.16 Семантические проверки

Семантические проверки конструкции языка программирования GAS-2023:

1. Наличие функции main
2. Единственность точки входа
3. Переопределение идентификаторов
4. Соответствие сигнатуры функции
5. Использование необъявленного идентификатора
6. Соответствие параметров функции
7. Совместимость типов
8. Правильность составленного оператора условия

1.17 Распределение оперативной памяти на этапе выполнения

Транслированный код использует две области памяти. В сегмент констант заносятся все литералы. В сегмент данных заносятся переменные и

параметры функций. Локальная область видимости в исходном коде определяется за счет использования правил именования идентификаторов и регулируется их префиксами, что и обуславливает их локальность на уровне исходного кода, несмотря на то, что в оттранслированном в язык ассемблера коде переменные имеют глобальную область видимости.

1.18 Стандартная библиотека и её состав

В языке CAS-2023 присутствует стандартная библиотека, которая подключается автоматически при трансляции исходного кода в язык ассемблера. Все стандартные библиотеки реализованы на языке C++. Стандартные библиотеки подключены по умолчанию в программу. Также в стандартной библиотеке реализованы функции для манипулирования выводом, недоступные конечному пользователю. Содержимое библиотеки и описание функций представлено в таблице 1.8.

Таблица 1.8 – Стандартная библиотека языка GAS-2023

Функция	Описание
<code>uint MathPow();</code>	Входной параметр: два параметра типа <code>uint</code> . Выходной параметр: возвращает число возведенное в степень Числовая функция, возвращает число, возведенное в степень.
<code>uint MathRand();</code>	Входной параметр: два параметра типа <code>uint</code> . Выходной параметр: возвращает любое число из заданного диапазона чисел Числовая функция, возвращает любое число из заданного диапазона.

При использовании языка GAS-2023, стандартные функции могут быть вызваны так же, как и пользовательские функции.

1.19 Ввод и вывод данных

Вывод данных осуществляется с помощью операторов `write` и `writeline`. Допускается их использование с литералами и идентификаторами.

В языке программирования GAS-2023 ввод данных не поддерживается.

Функции, управляющие вводом данных, реализованы на языке C++ и вызываются из транслированного кода, конечному пользователю недоступны. Пользовательская команды `write` и `writeline` в транслированном коде будут заменена вызовом нужных библиотечных функций. Библиотека, содержащая нужные процедуры, подключается на этапе генерации кода автоматически.

1.20 Точка входа

Точкой входа является функция `main`.

1.21 Препроцессор

В языке программирования GAS-20233 препроцессор не предусматривается.

1.22 Соглашения о вызовах

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

- все параметры функции передаются через стек;
- память высвобождает вызываемый код;
- занесение в стек параметров идёт справа налево.

1.23 Объектный код

Язык программирования GAS-2023 транслируется в язык ассемблера.

1.24 Классификация сообщений транслятора

В случае возникновения ошибки в исходном коде программы на языке программирования GAS-2023 и выявлении её транслятором в файл протокола выводится сообщение. Классификация обрабатываемых ошибок приведена в таблице 1.9.

Таблица 1.9 – Классификация сообщений транслятора

Интервал	Описание ошибок
0-99	Системные ошибки
100-119	Ошибки при работе с файлами
200-219	Ошибки лексического анализа
600-619	Ошибки синтаксического анализа
300-329	Ошибки семантического анализа

Каждый интервал ошибок соответствует определенной категории, такой как системные ошибки, ошибки работы с файлами, лексические ошибки, синтаксические ошибки и семантические ошибки.

1.25 Контрольный пример

Контрольный пример представлен в приложении А.

2 Структура транслятора

2.1 Компоненты транслятора, их назначение и принципы взаимодействия

В языке GAS-2023 исходный код транслируется в язык Assembler. Для того, чтобы получить ассемблерный код, используются выходные данные работы лексического анализатора, а именно таблица лексем и таблица идентификаторов. Для указания выходных файлов используются входные параметры транслятора, которые описаны в таблице 2.1. Структура транслятора языка GAS-2023 приведена на рисунке 2.1.

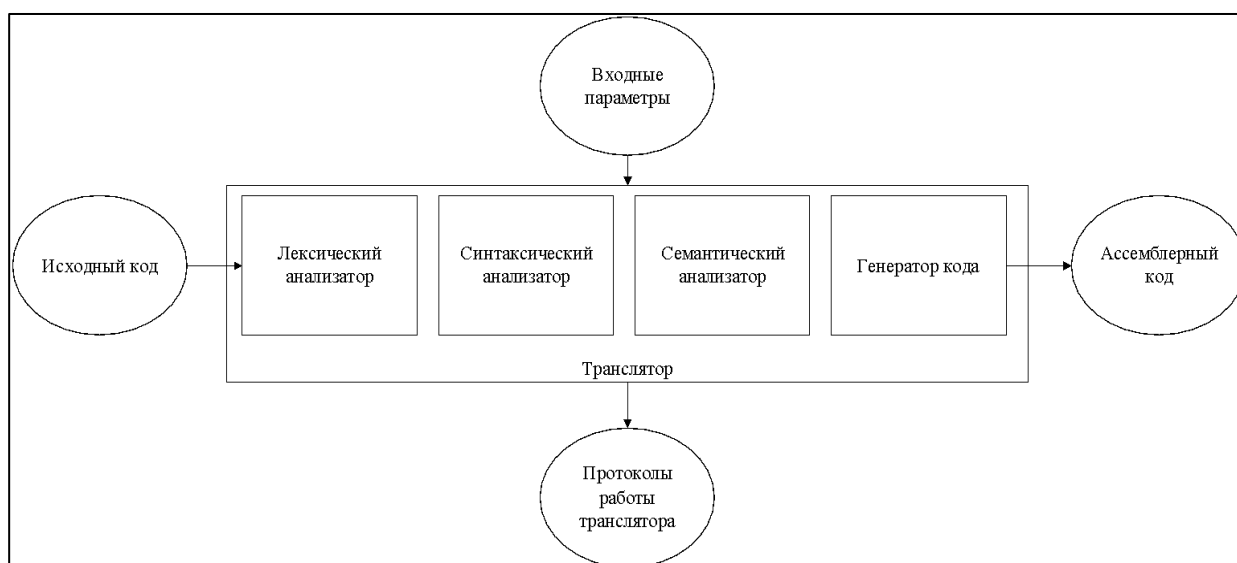


Рисунок 2.1 – Структура транслятора языка программирования GAS-2023

Транслятор разделен на несколько частей: лексический анализатор, синтаксический анализатор, семантический анализатор и генератор кода. Первая стадия работы компилятора называется лексическим анализом, а программа, её реализующая, – лексическим анализатором.

Цели лексического анализатора:

- убрать все лишние пробелы;
- выполнить распознавание лексем;
- построить таблицу лексем и таблицу идентификаторов;
- при неуспешном распознавании или обнаружении некоторых ошибок во входном тексте выдать сообщение об ошибке.

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

Семантический анализатор – часть транслятора, выполняющая семантический анализ, то есть проверку исходного кода на наличие ошибок, которые невозможно отследить при помощи регулярной и контекстно-

свободной грамматики. Входными данными являются таблица лексем и идентификаторов.

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

2.2 Перечень входных параметров транслятора

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

Таблица 2.1 – Входные параметры транслятора языка GAS-2023

Входной параметр	Описание параметра	Значение по умолчанию
-in:<путь к in-файлу>	Файл с исходным кодом на языке GAS-2023, имеющий расширение .txt	Не предусмотрено
-log:<путь к log-файлу>	Файл журнала для вывода протоколов работы программы	Значение по умолчанию: <имя in-файла>.log
-out:<путь к out-файлу>	Код программы сгенерированный на языке ассемблера	Значение по умолчанию: <имя in-файла>.asm

Входные параметры транслятора языка GAS-2023, указанные в таблице 2.1, определяют файлы, используемые для ввода и вывода результатов работы компонентов транслятора. Параметр "-in" указывает путь к файлу с исходным кодом на языке GAS-2023, "-log" определяет путь к файлу журнала для вывода протоколов работы программы, а "-out" задает путь к файлу, в котором будет сгенерирован ассемблерный код. По умолчанию, если не указаны конкретные значения, используются имена файлов, основанные на имени входного файла.

2.3 Перечень протоколов, формируемых транслятором и их содержимое

В ходе работы программы формируются протоколы работы лексического, синтаксического и семантического анализаторов, которые содержат в себе перечень протоколов работы. Также протокол содержит сообщения об ошибках на разных этапах компиляции. В таблице 2.2 приведены протоколы, формируемые транслятором и их содержимое.

Таблица 2.2 – Протоколы, формируемые транслятором языка GAS-2023

Формируемый протокол	Описание выходного протокола
Файл журнала, заданный параметром "-log:"	Содержит таблицу лексем и таблицу идентификаторов, протокол работы синтаксического анализатора и дерево разбора, полученные на этапе лексического и синтаксического анализа, результат работы алгоритма преобразования выражений к польской записи.
Выходной файл, с расширением "-asm:"	Результат работы программы – файл, содержащий исходный код на языке ассемблера.

Данная таблица помогает ознакомиться с протоколами, которые сформировались в ходе работы нашего компилятора.

3 Разработка лексического анализатора

3.1 Структура лексического анализатора

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

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

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

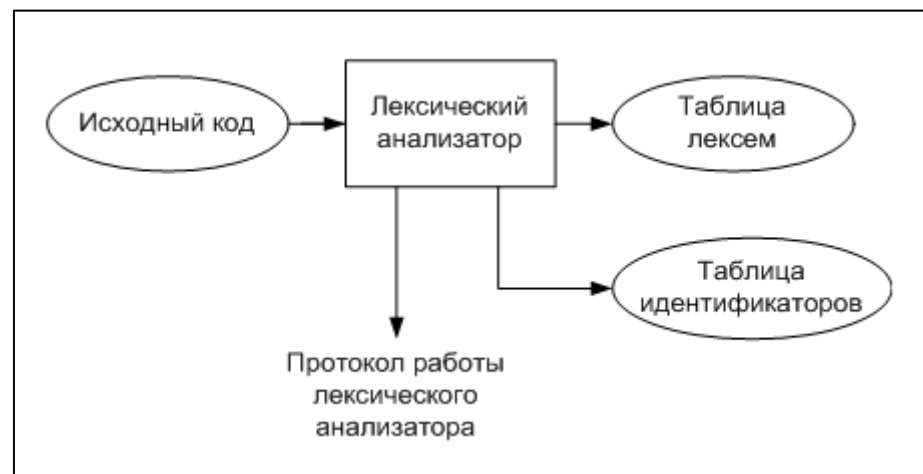


Рисунок 3.1 Структура лексического анализатора

Функции лексического анализатора:

- удаление «пустых» символов и комментариев. Для облегчения работы синтаксического анализатора
- распознавание идентификаторов и ключевых слов;
- распознавание разделителей и знаков операций.

3.2. Входные и выходные данные лексического анализатора

Лексический анализатор получает исходный код программы или текстовый файл в качестве входных данных. Он сканирует этот код, выделяет лексические единицы, такие как ключевые слова, операторы, идентификаторы, числа и другие символы.

Затем анализатор создает таблицы лексем и идентификаторов. Таблица лексем содержит информацию о каждой лексеме, включая ее тип, позицию в исходном коде и другие детали. Таблица идентификаторов содержит информацию о каждом обнаруженном идентификаторе, такую как его имя, тип и значение. Также может быть создана таблица сообщений, где записываются обнаруженные ошибки или предупреждения.

Полученные таблицы могут быть использованы для дальнейшего анализа и обработки исходного кода.

3.3 Параметры лексического анализатора

Входным параметром лексического анализатора является исходный текст программы, написанный на языке GAS-2023, а также файл протокола.

3.4 Алгоритм лексического анализа

- 1) Проверяет входной поток символов программы на исходном языке на допустимость, удаляет лишние пробелы и добавляет сепаратор для вычисления номера строки для каждой лексемы.
- 2) Для выделенной части входного потока выполняется функция распознавания лексемы.
- 3) При успешном распознавании информация о выделенной лексеме заносится в таблицу лексем и таблицу идентификаторов, и алгоритм возвращается к первому этапу.
- 4) Формирует протокол работы.
- 5) При неуспешном распознавании выдается сообщение об ошибке.

3.5. Контроль входных символов

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

```
#define IN_CODE_TABLE {\n
IN::F, IN::F, IN::F, IN::F, IN::F, IN::F, IN::F, IN::F, IN::F, IN::F, IN::T, IN::T, IN::T, IN::T, IN::T, IN::T,\n
IN::F, IN::F, IN::F, IN::F, IN::F, IN::F, IN::F, IN::F, IN::F, IN::F, IN::F, IN::F, IN::F, IN::F, IN::F, IN::F,\n
IN::T, IN::T, IN::T, IN::T, IN::T, IN::T, IN::T, IN::T, IN::T, IN::T, IN::T, IN::T, IN::T, IN::T, IN::T, IN::T,\n
IN::T, IN::T, IN::T, IN::T, IN::T, IN::T, IN::T, IN::T, IN::T, IN::T, IN::T, IN::T, IN::T, IN::T, IN::T, IN::T,\n
IN::T, IN::T, IN::T, IN::T, IN::T, IN::T, IN::T, IN::T, IN::T, IN::T, IN::T, IN::T, IN::T, IN::T, IN::T, IN::T,\n
IN::T, IN::T, IN::T, IN::T, IN::T, IN::T, IN::T, IN::T, IN::T, IN::T, IN::T, IN::T, IN::T, IN::T, IN::T, IN::T,\n
IN::T, IN::T, IN::T, IN::T, IN::T, IN::T, IN::T, IN::T, IN::T, IN::T, IN::T, IN::T, IN::T, IN::T, IN::T, IN::T,\n
IN::F, IN::F, IN::F, IN::F, IN::F, IN::F, IN::F, IN::F, IN::F, IN::F, IN::F, IN::F, IN::F, IN::F, IN::F, IN::F,\n
IN::F, IN::F, IN::F, IN::F, IN::F, IN::F, IN::F, IN::F, IN::F, IN::F, IN::F, IN::F, IN::F, IN::F, IN::F, IN::F,\n
IN::F, IN::F, IN::F, IN::F, IN::F, IN::F, IN::F, IN::F, IN::F, IN::F, IN::F, IN::F, IN::F, IN::F, IN::F, IN::F,\n
IN::T, IN::T, IN::T, IN::T, IN::T, IN::T, IN::T, IN::T, IN::T, IN::T, IN::T, IN::T, IN::T, IN::T, IN::T, IN::T,\n
IN::T, IN::T, IN::T, IN::T, IN::T, IN::T, IN::T, IN::T, IN::T, IN::T, IN::T, IN::T, IN::T, IN::T, IN::T, IN::T,\n
IN::T, IN::T, IN::T, IN::T, IN::T, IN::T, IN::T, IN::T, IN::T, IN::T, IN::T, IN::T, IN::T, IN::T, IN::T, IN::T,\n
IN::T, IN::T, IN::T, IN::T, IN::T, IN::T, IN::T, IN::T, IN::T, IN::T, IN::T, IN::T, IN::T, IN::T, IN::T, IN::T,\n
}
```

Рисунок 3.2. Таблица контроля входных символов

Контроль входных символов включает их идентификацию, классификацию и обработку соответствующим образом в лексическом анализаторе.

Таблица 3.1 – Соответствие символов и их значений в таблице

Значение в таблице входных символов	Символы
Разрешенный	T
Запрещенный	F
Игнорируемый	I

Это позволяет последующим компонентам транслятора правильно интерпретировать каждый символ и использовать их для дальнейшего анализа и генерации кода.

3.6 Удаление избыточных символов

Избыточным символом является символ пробела. Избыточный символ удаляется на этапе разбиения исходного кода на токены.

Описание алгоритма удаления избыточного символа:

1. Посимвольно считываем файл с исходным кодом программы;
2. Встреча пробела является своего рода встречей символа-сепаратора;
3. В отличие от других символов-сепараторов не записываем в очередь лексем пробел.

3.7 Перечень ключевых слов

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

Таблица 3.2 – Соответствие токенов и сепараторов с лексемами

Токен	Лексема	Пояснение
uint, str, bool	t	Названия типов данных языка.
Идентификатор	i	Длина идентификатора – 10 символов.
Литерал	l	Литерал любого доступного типа.
method	f	Объявление функции.
push	r	Выход из функции.
main	m	Главная функция.
create	d	Объявление переменной.
if	?	Объявление условия
else	e	Объявления условия

Окончание таблицы 3.2

Токен	Лексема	Пояснение
write	p	Вывод в консоль
writeline	n	Вывод на консоль с переводом строки
;	;	Разделение выражений.
,	,	Разделение параметров функций.
((Передача параметров в функцию, приоритет операций.
))	Закрытие блока для передачи параметров, приоритет операций.
{	{	Открытие тела функции или цикла
=	=	Знак присваивания.
+, -	v	Знаки арифметических операций
left, right	v	Сдвиговые операции
< > & !	v	Операции сравнения

Пример реализации таблицы лексем представлен в приложении Б.

Каждому выражению соответствует детерминированный конечный автомат, по которому происходит разбор данного выражения. Структура конечного автомата и пример графа перехода конечного автомата изображены на рисунках 3.3 и 3.4 соответственно.

```

struct RELATION //ребро
{
    char symbol;    ///символ перехода
    short nnode;    ///номер инцидентной вершины
    RELATION(
        char c = 0x00,
        short ns = NULL
    );
};

struct NODE //вершина графа
{
    short n_relation;    ///количество инцидентных ребер
    RELATION *relations; ///инцидентные ребра
    NODE();
    NODE(
        short n,
        RELATION rel, ...
    );
};

struct FST //КА
{
    char *string;    ///цепочка(строка, завершается 0x00)
    short position;  ///текущая позиция в цепочке
    short nstates;   ///кол-во состояний автомата
    NODE *nodes;    ///граф переходов: [0] - начальное сост., [nstate-1] - конечное
    short *rstates;  ///возможные состояния автомата на данной позиции
    FST(
        char *s,
        short ns,
        NODE n, ...
    );
};

```

Рисунок 3.3 – Структура конечного автомата

На каждый автомат в массиве подаётся токен и с помощью регулярного выражения, соответствующего данному графу переходов, происходит разбор. В случае успешного разбора выражения оно записывается в таблицу лексем.

```

bool literaluint16(char word[]) {
    FST::FST literaluint16(word, 4,
        FST::NODE(1, FST::RELATION('1', 1)),
        FST::NODE(1, FST::RELATION('6', 2)),
        FST::NODE(1, FST::RELATION('x', 3)),
        FST::NODE(22,
            FST::RELATION('0', 3), FST::RELATION('1', 3), FST::RELATION('2', 3),
            FST::RELATION('3', 3), FST::RELATION('4', 3), FST::RELATION('5', 3),
            FST::RELATION('6', 3), FST::RELATION('7', 3), FST::RELATION('8', 3),
            FST::RELATION('9', 3), FST::RELATION('A', 3), FST::RELATION('B', 3),
            FST::RELATION('C', 3), FST::RELATION('D', 3), FST::RELATION('E', 3),
            FST::RELATION('F', 3), FST::RELATION('a', 3), FST::RELATION('b', 3),
            FST::RELATION('c', 3), FST::RELATION('d', 3), FST::RELATION('e', 3),
            FST::RELATION('f', 3)
        ),
        FST::NODE()
    );
    return FST::execute(literaluint16);
}

```

Рисунок 3.4 – Пример реализации графа КА для литерала

Если выражение является идентификатором или литералом, информация также заносится в таблицу идентификаторов.

3.8 Основные структуры данных

Основными структурами данных лексического анализатора являются таблица лексем и таблица идентификаторов. Код со структурой таблицы идентификаторов представлен на рисунке 3.5. Код со структурой таблицы лексем представлен на рисунке 3.6.

```

enum IDDATATYPE { UINT = 1, BOOL = 2, STR = 3 }; //типы данных идентифи
enum IDTYPE { V = 1, F = 2, P = 3, L = 4, OP = 5 }; //типы идентификаторов: пе
struct Entry //строка TI
{
    int idxfirstLE; //индекс первой строки в таблице лексем
    char id[ID_MAXSIZE * 2 + 1]; //идентификатор(автоматически усекается до
    IDDATATYPE iddatatype; //тип данных
    IDTYPE idtype; //тип идентификатора
    struct
    {
        int count;
        IDDATATYPE type[MAX_PARAMS];
    } params;
    struct //
    {
        unsigned int vuint = 0; // значение unsigned integer
        bool vbool;
        struct
        {
            int len; // количество символов в string
            char str[TI_STR_MAXSIZE - 1]; // символы string
        } vstr; // значение string
    } value; // значение идентификатора
};

struct IdTable //экземпляр TI
{
    int maxsize; //емкость TI < TI_MAXSIZE
    int size; //текущий размер TI < maxsize
    Entry* table; //массив строк TI
};

```

Рисунок 3.5 – Структура таблицы идентификаторов

Таблица идентификаторов содержит имя идентификатора (id), номер в таблице лексем (idxfirstLE), тип данных (iddatatype), тип идентификатора (idtype) и значение (или параметры функций) (value).

```

struct Entry //строка таблицы лексем
{
    char lexema; //лексема
    int sn; //номер строки в исх. тексте
    int idxTI; //индекс в TI или в LT_TI_NULLIDX
    int priority;
    operations op;
};

struct LexTable //экземпляр таблицы лексем
{
    int maxsize; //емкость ТЛ < LT_MAXSIZE
    int size; //тек. размер ТЛ < maxsize
    Entry* table; //массив строк ТЛ
};

```

Рисунок 3.6 – Структура таблицы лексем

Таблица лексем содержит номер лексемы, лексему (lexema), полученную при разборе, номер строки в исходном коде (line), номер в таблице идентификаторов, если лексема является идентификатором (idxTI) и приоритет, если лексема является операцией.

3.9 Структура и перечень сообщений лексического анализа

Ошибки, возникающие в процессе трансляции программы, фиксируются в протокол, заданный входными параметрами. В случае возникновения ошибок происходит их протоколирование с номером ошибки и диагностическим сообщением. Если в процессе анализа находятся более трёх ошибок, то анализ останавливается.

3.10 Принцип обработки ошибок

Для обработки ошибок лексический анализатор использует таблицу с сообщениями. Структура сообщений содержит информацию о номере сообщения, номере строки и позицию, где было вызвано сообщение в исходном коде, информацию об ошибке. Перечень сообщений представлен в таблице 3.3.

Таблица 3.3 – Перечень ошибок

Код ошибки	Значение ошибки
200	[Лексическая ошибка] Запрещенный символ в исходном файле (-in)
201	[Лексическая ошибка] Размер таблицы лексем превышен
203	[Лексическая ошибка] Размер таблицы идентификаторов превышен
205	[Лексическая ошибка] Неизвестная последовательность символов

При возникновении сообщения, лексический анализатор игнорирует найденную ошибку и продолжает работу с исходным кодом, при условии, что найденных ошибок не больше трех.

3.11 Контрольный пример

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

4 Разработка синтаксического анализатора

4.1 Структура синтаксического анализатора

Синтаксический анализатор: часть компилятора, выполняющая синтаксический анализ, то есть исходный код проверяется на соответствие правилам грамматики. Описание структуры синтаксического анализатора языка представлено на рисунке 4.1.

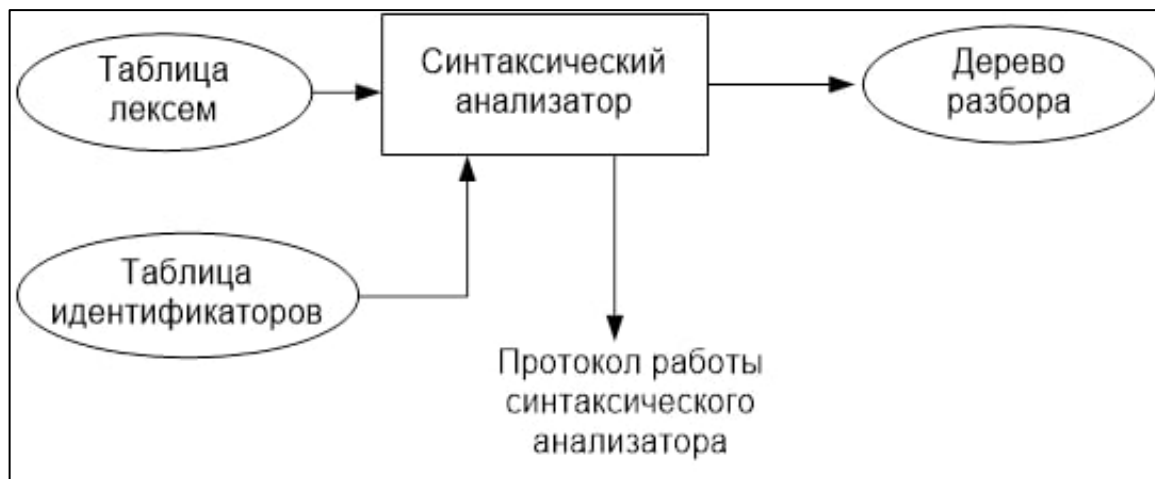


Рисунок 4.1 Структура синтаксического анализатора.

Входной информацией для синтаксического анализа является таблица лексем и таблица идентификаторов. Выходной информацией – дерево разбора.

4.2 Контекстно-свободная грамматика, описывающая синтаксис

В синтаксическом анализаторе транслятора языка GAS-2023 используется контекстно-свободная грамматика $G = \langle T, N, P, S \rangle$, 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 и правила P имеют вид:

1) $A \rightarrow a\alpha$, где $a \in T, \alpha \in (T \cup N) \cup \{\lambda\}$; (или $\alpha \in (T \cup N)$, или $\alpha \in V$)

2) $S \rightarrow \lambda$, где $S \in N$ – начальный символ, при этом если такое правило существует, то нетерминал S не встречается в правой части правил.

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

NS – нетерминальные символы, представленные несколькими заглавными буквами латинского алфавита. Описание нетерминальных символов содержится в таблице 4.1.

Таблица 4.1 – Описание нетерминальных символов

Нетерминал	Цепочки правил	Описание
S	tfiFBS m{N} tfiFB	Проверка правильности структуры программы
F	(P) ()	Проверка наличия параметров функции
P	ti ti,P	Проверка на правильность параметров функции при её объявлении
B	{NrI;} {rI;}	Проверка наличия тела функции
I	i l	Проверка на недопустимое выражение
N	dti;N dti=E;N i=E;N wI;N nI;N dti; dti=E; i=E; ?(R){X} ?(R){X}e{X} wI; nI;	Проверка на правильность конструкции в теле функции
R	i l ivi ivl lvi lvl	Проверка на правильность конструкции в условии оператора if
K	(W) ()	Проверка на правильность вызова функции
E	i iM lM (E) (E)M iK iKM	Проверка на правильность арифметического выражения
W	i l i,W l,W	Проверка на правильность параметров вызываемой функции

Окончание таблицы 4.1

Нетерминал	Цепочки правил	Описание
M	vE	Проверка на правильность арифметических действий
X	i=E;X nI;X wI;X i = E; nI; wI;	Проверка на правильность конструкции условного оператора

Правила языка GAS-2023 представлены в приложении В.

4.3 Построение конечного магазинного автомата

Конечный автомат с магазинной памятью представляет собой семёрку $M = \langle Q, V, Z, \delta, q_0, z_0, F \rangle$, описание которой представлено в таблице 4.2.

Таблица 4.2 – Описание компонентов магазинного автомата

Компонента	Определение	Описание
Q	Множество состояний автомата	Состояние автомата представляет из себя структуру, содержащую позицию на входной ленте, номера текущего правила и цепочки и стек автомата
V	Алфавит входных символов	Алфавит представляет из себя множества терминальных и нетерминальных символов, описание которых содержится в таблица 1.2 и 4.1.
Z	Алфавит специальных магазинных символов	Алфавит магазинных символов содержит стартовый символ и маркер дна стека (символ \$)
δ	Функция переходов автомата	Функция представляет из себя множество правил грамматики, описанных в таблице 4.1.
q_0	Начальное состояние автомата	Состояние, которое приобретает автомат в начале своей работы, в виде стартового правила грамматики (нетерминальный символ A)
z_0	Начальное состояние магазина автомата	Символ маркера дна стека (\$)
F	Множество конечных состояний	Конечные состояние заставляют автомат прекратить свою работу. Конечное состояние – пустой магазин автомата и совпадение позиции на входной ленте автомата с размером ленты

Структура данного автомата показана в приложении Г.

4.4 Основные структуры данных

Основные структуры данных синтаксического анализатора представляются в виде структуры магазинного конечного автомата, выполняющего разбор исходной ленты, и структуры грамматики Грейбах, описывающей синтаксические правила языка GAS-2023. Данные структуры в приложении В.

4.5 Описание алгоритма синтаксического разбора

Принцип работы автомата следующий:

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

4.6 Параметры синтаксического анализатора

Для синтаксического анализатора входными данными являются таблицы лексем и идентификаторов. Кроме того, используется описание грамматики, представленное в форме Грейбаха. Результаты лексического разбора, включая дерево разбора и протокол работы автомата с магазинной памятью, записываются в журнал работы программы.

4.7 Структура и перечень сообщений синтаксического анализатора

Перечень сообщений синтаксического анализатора представлен в таблице 4.3.

Таблица 4.3 – Перечень ошибок

Код ошибки	Значение ошибки
600	[Синтаксическая ошибка] Неверная структура программы
601	[Синтаксическая ошибка] Нет реализации main
602	[Синтаксическая ошибка] Ошибка в параметрах функции

Окончание таблицы 4.3

Код ошибки	Значение ошибки
603	[Синтаксическая ошибка] Отсутствует тело функции
604	[Синтаксическая ошибка] Недопустимое выражение
606	[Синтаксическая ошибка] Неверная конструкция в теле функции
607	[Синтаксическая ошибка] Ошибка в условном операторе
608	[Синтаксическая ошибка] Ошибка в вызове функции
610	[Синтаксическая ошибка] Ошибка в списке параметров при вызове функции
611	[Синтаксическая ошибка] Подробная информация в log файле

Код ошибки и значение ошибки предоставляют информацию о конкретных проблемах в синтаксисе программы. В случае возникновения ошибки, синтаксический анализатор может сгенерировать сообщение с соответствующим кодом ошибки и описанием.

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. Все ошибки записываются в общую структуру ошибок.

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 Г в виде фрагмента трассировки и дерева разбора исходного кода.

5 Разработка семантического анализатора

5.1 Структура семантического анализатора

Семантический анализатор принимает на свой вход результаты работ лексического и синтаксического анализаторов, то есть таблицы лексем, идентификаторов и результат работы синтаксического анализатора, то есть дерево разбора, и последовательно ищет необходимые ошибки. Общая структура обособленно работающего (не параллельно с лексическим анализом) семантического анализатора представлена на рисунке 5.1.



Рисунок 5.1. Структура семантического анализатора

Некоторые проверки (такие как проверка на единственность точки входа, проверка на предварительное объявление переменной) осуществляются в процессе лексического анализа.

5.2 Функции семантического анализатора

Семантический анализатор проверяет правильность составления программных конструкций. При обнаружении ошибки будет выведен код ошибки, а также информация о данной ошибке. Информация об ошибках выводится в консоль, а также в протокол работы.

5.3 Структура и перечень семантических ошибок

Сообщения, формируемые семантическим анализатором, представлены на рисунке 5.2.

Таблица 5.1 – Перечень ошибок

Код ошибки	Значение ошибки
301	[Семантическая ошибка] Имеется более одной точки входа в main
302	[Семантическая ошибка] Не имеется точки входа в main
303	[Семантическая ошибка] Объявление глобальной переменной
304	[Семантическая ошибка] Объявление переменной без ключевого слова create

Окончание таблицы 5.1

305	[Семантическая ошибка] Необъявленный идентификатор
306	[Семантическая ошибка] Объявление переменной без указания типа
307	[Семантическая ошибка] Попытка реализовать существующую функцию
308	[Семантическая ошибка] Повторное объявление идентификатора
309	[Семантическая ошибка] Несовпадение типов передаваемых параметров функции
310	[Семантическая ошибка] Несоответствие арифметических операторов
311	[Семантическая ошибка] Неправильный параметр в условном операторе
312	[Семантическая ошибка] Несоответствие типов данных
313	[Семантическая ошибка] Значение uint превышено
314	[Семантическая ошибка] Функция возвращает неверный тип данных
315	[Семантическая ошибка] Несоответствие количества передаваемых параметров функции
316	[Семантическая ошибка] Длина строки превышена
317	[Семантическая ошибка] Длина идентификатора превышена
318	[Семантическая ошибка] Превышено число параметров функции
320	[Семантическая ошибка] Использование рекурсии
321	[Семантическая ошибка] Использование нулевой строки

Код ошибки и значение ошибки предоставляют информацию о конкретных проблемах в семантике программы. В случае возникновения ошибки, синтаксический анализатор может сгенерировать сообщение с соответствующим кодом ошибки и описанием.

5.4 Принцип обработки ошибок

Ошибки, возникающие в процессе трансляции программы, фиксируются в протокол, заданный входным параметрами. В случае возникновения ошибок происходит их протоколирование с номером ошибки и диагностическим сообщением. Анализ останавливается после того, как будут найдены все ошибки. После завершения процесса трансляции и протоколирования ошибок, программист может проанализировать протокол, найти все ошибки и приступить к их исправлению.

5.5 Контрольный пример

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

Таблица 5.1 – Примеры диагностики ошибок

Исходный код	Текст сообщения
<pre>{ create uint x; create uint y; y = 10; x = y - 50; }</pre>	<p>Ошибка 302: [Семантическая ошибка] Не имеется точки входа в main</p>
<pre>main { create uint x; create str string; string = x; }</pre>	<p>Ошибка 312: [Семантическая ошибка] Несоответствие типов данных Строка 5 позиция 9</p>
<pre>uint method same() { push 0; } uint method same() { push 1; } main { create uint result = same(); }</pre>	<p>Ошибка 307: [Семантическая ошибка] Попытка реализовать существующую функцию Строка 6 позиция 12</p>

Каждый пример содержит фрагмент исходного кода, в котором допущена ошибка, и текст сообщения, который описывает эту ошибку.

6 Вычисление выражений

6.1 Выражения, допускаемые языком

В языке GAS-2023 допускаются выражения, применимые к целочисленным типам данных. В выражениях поддерживаются арифметические операции, такие как +, -, сдвиговые операции, как left, right и (), а также вызовы функций как операнды арифметических выражений.

Приоритет операций представлен в таблице 6.1.

Таблица 6.1 – Приоритет операций в языке GAS-2023

Приоритет	Операция
1	+ -
2	left right

Правильное понимание приоритета операций в выражениях помогает корректно оценивать порядок выполнения операций и получать ожидаемые результаты при вычислении выражений в языке GAS-2023.

6.2 Польская запись и принцип ее построения

Все выражения языка GAS-2023 преобразовываются к обратной польской записи.

Польская запись - это альтернативный способ представления арифметических выражений, в котором операторы располагаются перед или после своих операндов, в отличие от классической инфиксной записи, где операторы находятся между операндами. Польская запись может быть прямой (префиксной) или обратной (постфиксной).

Таблица 6.2 – Пример преобразования выражения в обратную польскую запись

Исходная строка	Результирующая строка	Стек
m-func(m)		
-func(m)	m	
func(m)	m	-
(m)	m	-
m)	m	-
)	mm	-
	mm@ 1-	

Алгоритм построения польской записи:

- исходная строка: выражение;
- результирующая строка: польская запись;
- стек: пустой;
- исходная строка просматривается слева направо;
- операнды переносятся в результирующую строку;
- операция записывается в стек, если стек пуст;

- операция выталкивает все операции с большим или равным приоритетом в результирующую строку;
- отрывающая скобка помещается в стек;
- закрывающая скобка выталкивает все операции;

6.3 Программная реализация обработки выражений

Программная реализация функции перевода в обратную польскую инверсию содержится в функции `StartPolish`, которая принимает параметром таблицу лексем. Она содержит цикл, который при нахождении символа присваивания (=) вызывает функцию `PolishNotation` и преобразует последующее выражение до конца строки.

После завершения функции `StartPolish` происходит синхронизация индексов таблицы идентификаторов с таблицей лексем, так как лексемы меняют свое положение.

6.4 Контрольный пример

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

7 Генерация кода

7.1 Структура генератора кода

Генерация объектного кода — это перевод компилятором внутреннего представления исходной программы в цепочку символов выходного языка.



Рисунок 7.1 – Структура генератора кода

На вход генератора подаются таблицы лексем и идентификаторов, на основе которых генерируется файл с ассемблерным кодом.

7.2 Представление типов данных в оперативной памяти

Элементы таблицы идентификаторов расположены сегментах `.data` и `.const` языка ассемблера. Соответствия между типами данных идентификаторов на языке GAS-2023 и на языке ассемблера приведены в таблице 7.1.

Таблица 7.1 – Соответствия типов идентификаторов языка и языка ассемблера

Тип идентификатора на языке GAS-2023	Тип идентификатора на языке ассемблера	Пояснение
uint	dword	Хранит целочисленный тип данных без знака.
bool	dword	Хранит целочисленный тип данных без знака.
str	byte	Каждый символ строки типа str хранится в поле размером 1 байт.

Каждый тип идентификатора в GAS-2023 имеет соответствующий тип идентификатора в языке ассемблера, которые указывают на способ представления данных в памяти.

7.3 Статическая библиотека

Статическая библиотека реализована на языке программирования C++.

Её реализация находится в проекте StaticLib, в свойствах которого был выбран пункт «статическая библиотека .lib».

В языке программирования GAS-2023, библиотеки подключаются по умолчанию. Для подключения библиотеки в ассемблерном языке используется директива `includelib` на этапе генерации кода. Затем, с помощью оператора `EXTRN`, объявляются имена функций из подключенной библиотеки. Оператор `EXTRN` выполняет две функции. Во-первых, он информирует ассемблер о том, что указанное символическое имя является внешним для текущего ассемблирования. Во-вторых, оператор `EXTRN` указывает ассемблеру тип соответствующего символического имени. Приведенный выше процесс иллюстрируется на листинге 7.1.

```
.586
.model flat, stdcall
includelib libcrt.lib
includelib kernel32.lib
includelib ../Debug/StaticLib.lib
ExitProcess PROTO :DWORD

MathPow PROTO :DWORD, :DWORD
MathRand PROTO :DWORD, :DWORD
OutputStr PROTO :DWORD
OutputStrNoLine PROTO :DWORD
OutputInt PROTO :DWORD
OutputIntNoLine PROTO :DWORD
```

Листинг 7.1 Фрагмент функции генерации кода

Это необходимо, так как ассемблеру нужно знать, как интерпретировать каждый символ для правильной генерации команд.

7.4 Особенности алгоритма генерации кода

В языке GAS-2023 генерация кода строится на основе таблиц лексем и идентификаторов. Общая схема работы генератора кода представлена на рисунке 7.2.

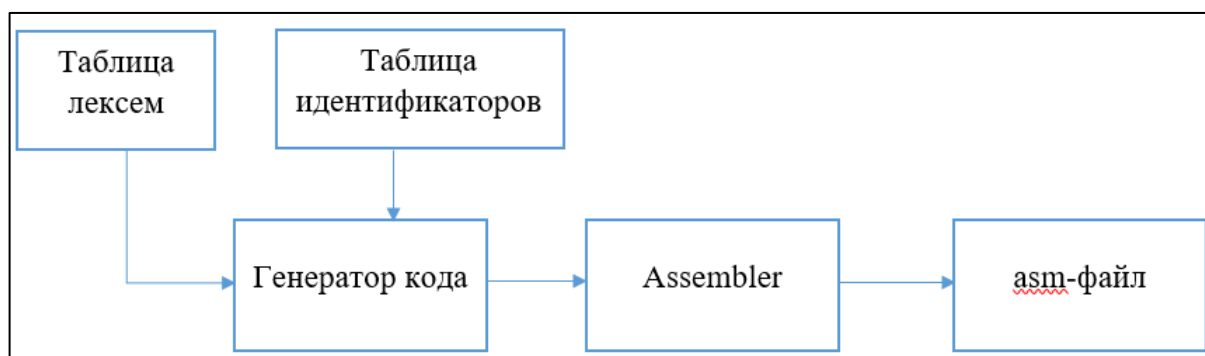


Рисунок 7.1 – Структура генератора кода

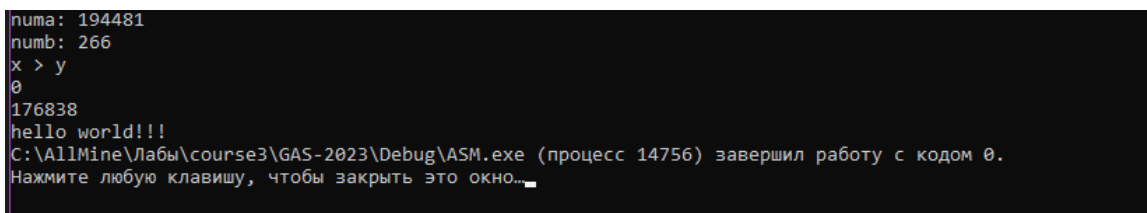
Алгоритм генерации кода в GAS-2023 основан на таблицах лексем и идентификаторов. Эти таблицы содержат информацию о различных лексемах и идентификаторах, которые используются в программе.

7.5 Входные параметры генератора кода

На вход генератору кода поступают таблицы лексем и идентификаторов исходного код программы на языке GAS-2023. Результаты работы генератора кода выводятся в файл с расширением .asm.

7.6 Контрольный пример

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



```
numa: 194481
numb: 266
x > y
0
176838
hello world!!!
C:\AllMine\Лабы\course3\GAS-2023\Debug\ASM.exe (процесс 14756) завершил работу с кодом 0.
Нажмите любую клавишу, чтобы закрыть это окно...
```

Рисунок 7.2 Результат работы программы на языке GAS-2023

На консоли отображается вывод строк, чисел и булевского значения. За вывод на консоль каких-либо значений происходит с помощью функций статической библиотеки.

8 Тестирование транслятора

8.1 Общие положение

Для эффективного тестирования компилятора используются принципы:

- 1) Тестирование должно охватывать все основные функции и возможности компилятора.
- 2) Тестовые данные должны отражать типичные ситуации, с которыми компилятор может столкнуться в реальных условиях. Это включает использование различных комбинаций символов, операторов, ключевых слов и других конструкций языка.
- 3) Тестирование должно проверять поведение компилятора в различных ситуациях, например, обработку файлов максимального размера или использование максимально возможного количества переменных.

При обнаружении ошибки во время компиляции, компилятор должен выполнять следующие действия:

- 1) Предоставить информативное сообщение об ошибке, указывающее ее тип, местоположение в коде и предложение по исправлению.
- 2) Записать результаты тестирования, включая обнаруженные ошибки, в лог-файл (.log). Протокол должен содержать подробную информацию о каждом тесте, включая входные данные, ожидаемые результаты и фактические результаты.

8.2 Результаты тестирования

В языке программирования GAS-2023 не разрешается использовать запрещенные входным алфавитом символы. Результат использования запрещенного символа показан в таблице 8.1.

Таблица 8.1 – Тестирование фазы проверки на допустимость символов

Исходный код	Диагностическое сообщение
<code>uint method ёfeev{ret 11;}</code>	Ошибка 111: [Лексическая ошибка] Недопустимый символ в исходном файле (-in) Строка 1 позиция 9

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
<code>main { create uint \$adv; }</code>	Ошибка 205: [Лексическая ошибка] Неизвестная последовательность символов Строка 3 позиция 12

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
<pre>uint method func(str s, uint x) { push x; } main { create uint res = func(9,); }</pre>	<p>Строка 8, [Синтаксическая ошибка] Ошибка в списке параметров при вызове функции</p>

Результаты тестирования показали, что синтаксический анализатор успешно выполняет свою функцию.

Заключение

В ходе выполнения курсовой работы был разработан транслятор и генератор кода для языка программирования GAS-2023 со всеми необходимыми компонентами. Таким образом, были выполнены основные задачи данной курсовой работы:

1. Сформулирована спецификация языка GAS-2023;
2. Разработаны конечные автоматы и важные алгоритмы на их основе для эффективной работы лексического анализатора;
3. Осуществлена программная реализация лексического анализатора, распознающего допустимые цепочки спроектированного языка;
4. Разработана контекстно-свободная, приведённая к нормальной форме Грейбах, грамматика для описания синтаксически верных конструкций языка;
5. Осуществлена программная реализация синтаксического анализатора;
6. Разработан семантический анализатор, осуществляющий проверку используемых инструкций на соответствие логическим правилам;
7. Разработан транслятор кода на язык ассемблера;
8. Проведено тестирование всех вышеперечисленных компонентов.

Окончательная версия языка GAS-2023 включает:

1. 3 типа данных;
2. Поддержка операторов вывода и перевода строки;
3. Наличие 4 арифметических операторов для вычисления выражений
4. Наличие 2 сдвиговых операций
5. Наличие 4 операций сравнения
6. Поддержка функций и условных операторов
7. Наличие библиотеки стандартных функций языка
8. Структурированная и классифицированная система для обработки ошибок пользователя.

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

Список использованных источников

1. Ахо, А. Компиляторы: принципы, технологии и инструменты / А. Ахо, Р. Сети, Дж. Ульман. – М.: Вильямс, 2003. – 768с.
2. Герберт, Ш. Справочник программиста по С/С++ / Шилдт Герберт. - 3-е изд. – Москва : Вильямс, 2003. - 429 с.
3. Прата, С. Язык программирования С++. Лекции и упражнения / С. Прата. – М., 2006 — 1104 с.
4. Страуструп, Б. Принципы и практика использования С++ / Б. Страуструп – 2009 – 1238 с

Приложение А

Листинг 1 Исходный код программы на языке GAS-2023

```
uint method change(uint a, uint b)
{
    if(a & b){
        a = b left 4;
    } else {
        a = (a left 0b11) + b;
    }
    push a;
}

bool method checknums(uint x, uint y)
{
    if(x > y)
    {
        writeline 'x > y';
    } else {
        writeline 'x < y';
    }
    push false;
}

main {
    create uint numa = MathPow(0b10101, 0b100);
    create uint numb = MathRand(16xb1, 16x15a);
    write 'numa: ';
    writeline numa;
    write 'numb: ';
    writeline numb;
    create bool res = checknums(numa, numb);
    writeline res;
    create uint i = 0b1111;
    create uint y = ((i right 2) + change(16x5678, 0b1101)) - numb;
    writeline y;
    create str text = 'hello world!!!';
    write text;
}
```


Приложение Б

Листинг 1 Таблица идентификаторов контрольного примера

#	Идентификатор	Тип данных	Тип идентификатора	Значение

0000	change	uint	функция	-
0001	change_a	uint	параметр	-
0002	change_b	uint	параметр	-
0003	&	-	оператор	-
0004	:	-	оператор	-
0005	L1	uint	литерал	4
0006	L2	uint	литерал	3
0007	+	-	оператор	-
0008	checknums	bool	функция	-
0009	checknums_x	uint	параметр	-
0010	checknums_y	uint	параметр	-
0011	>	-	оператор	-
0012	L3	str	литерал	[5]"x > y"
0013	L4	str	литерал	[5]"x < y"
0014	L5	bool	литерал	0
0015	main	uint	функция	-
0016	main_numa	uint	переменная	0
0017	MathPow	uint	функция	-
0018	L6	uint	литерал	21
0019	main_numb	uint	переменная	0
0020	MathRand	uint	функция	-
0021	L7	uint	литерал	177
0022	L8	uint	литерал	346
0023	L9	str	литерал	[6]"numa: "
0024	L10	str	литерал	[6]"numb: "
0025	main_res	bool	переменная	0
0026	main_i	uint	переменная	0
0027	L11	uint	литерал	15
0028	main_y	uint	переменная	0
0029	^	-	оператор	-
0030	L12	uint	литерал	2
0031	L13	uint	литерал	22136
0032	L14	uint	литерал	13
0033	-	-	оператор	-
0034	main_text	str	переменная	[0]""
0035	L15	str	литерал	[14]"hello world!!!"

Количество идентификаторов: 36				

Листинг 2 Таблица лексем контрольного примера

#	Лексема	Строка	
Индекс в ТИ			

0000	t		1	-
0001	f		1	-
0002	i		1	0
0003	(1	-
0004	t		1	-
0005	i		1	1
0006	,		1	-
0007	t		1	-
0008	i		1	2
0009)		1	-
0010	{		2	-
0011	?		3	-
0012	(3	-
0013	i		3	1
0014	v		3	3
0015	i		3	2
0016)		3	-
0017	{		3	-
0018	i		4	1
0019	=		4	-
0020	i		4	2
0021	v	:	4	4
0022	l		4	5
0023	;		4	-
0024	}		5	-
0025	e		5	-
0026	{		5	-
0027	i		6	1
0028	=		6	-
0029	(6	-
0030	i		6	1
0031	v	:	6	4
0032	l		6	6
0033)		6	-
0034	v	+	6	7
0035	i		6	2
0036	;		6	-
0037	}		7	-
0038	r		8	-
0039	i		8	1
0040	;		8	-
0041	}		9	-
0042	t		11	-
0043	f		11	-
0044	i		11	8
0045	(11	-
0046	t		11	-
0047	i		11	9
0048	,		11	-
0049	t		11	-
0050	i		11	10

0051)		11		-
0052		{		12		-
0053		?		13		-
0054		(13		-
0055		i		13		9
0056		v		13		11
0057		i		13		10
0058)		13		-
0059		{		14		-
0060		n		15		-
0061		l		15		12
0062		;		15		-
0063		}		16		-
0064		e		16		-
0065		{		16		-
0066		n		17		-
0067		l		17		13
0068		;		17		-
0069		}		18		-
0070		r		19		-
0071		l		19		14
0072		;		19		-
0073		}		20		-
0074		m		22		15
0075		{		22		-
0076		d		23		-
0077		t		23		-
0078		i		23		16
0079		=		23		-
0080		i		23		17
0081		(23		-
0082		l		23		18
0083		,		23		-
0084		l		23		5
0085)		23		-
0086		;		23		-
0087		d		24		-
0088		t		24		-
0089		i		24		19
0090		=		24		-
0091		i		24		20
0092		(24		-
0093		l		24		21
0094		,		24		-
0095		l		24		22
0096)		24		-
0097		;		24		-
0098		w		25		-
0099		l		25		23
0100		;		25		-
0101		n		26		-
0102		i		26		16
0103		;		26		-

0104		w			27			-
0105		l			27			24
0106		;			27			-
0107		n			28			-
0108		i			28			19
0109		;			28			-
0110		d			29			-
0111		t			29			-
0112		i			29			25
0113		=			29			-
0114		i			29			8
0115		(29			-
0116		i			29			16
0117		,			29			-
0118		i			29			19
0119)			29			-
0120		;			29			-
0121		n			30			-
0122		i			30			25
0123		;			30			-
0124		d			31			-
0125		t			31			-
0126		i			31			26
0127		=			31			-
0128		l			31			27
0129		;			31			-
0130		d			32			-
0131		t			32			-
0132		i			32			28
0133		=			32			-
0134		(32			-
0135		(32			-
0136		i			32			26
0137		v	^		32			29
0138		l			32			30
0139)			32			-
0140		v	+		32			7
0141		i			32			0
0142		(32			-
0143		l			32			31
0144		,			32			-
0145		l			32			32
0146)			32			-
0147)			32			-
0148		v	-		32			33
0149		i			32			19
0150		;			32			-
0151		n			33			-
0152		i			33			28
0153		;			33			-
0154		d			34			-
0155		t			34			-
0156		i			34			34

0157		=		34		-
0158		l		34		35
0159		;		34		-
0160		w		35		-
0161		i		35		34
0162		;		35		-
0163		}		36		-

Всего лексем: 164

Приложение В

Листинг 1 – Правила языка GAS-2023

```

namespace GRB
{
#define NS(n) Rule::Chain::N(n)
#define TS(n) Rule::Chain::T(n)

    Greibach greibach(
        NS('S'),
        TS('$'),
        11,
        Rule(NS('S'), GRB_ERROR_SERIES + 0,
            3,
            Rule::Chain(4, TS('m'), TS('{'), NS('N'),
TS('{}')),
            Rule::Chain(6, TS('t'), TS('f'), TS('i'),
NS('F'), NS('B'), NS('S')),
            Rule::Chain(5, TS('t'), TS('f'), TS('i'),
NS('F'), NS('B'))
        ),
        Rule(
            NS('F'), GRB_ERROR_SERIES + 2,
            2,
            Rule::Chain(3, TS('('), NS('P'), TS(')')),
            Rule::Chain(2, TS('('), TS(')'))
        ),
        Rule(
            NS('P'), GRB_ERROR_SERIES + 2,
            2,
            Rule::Chain(2, TS('t'), TS('i')),
            Rule::Chain(4, TS('t'), TS('i'), TS(','),
NS('P'))
        ),
        Rule(
            NS('B'), GRB_ERROR_SERIES + 3,
            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('I'), GRB_ERROR_SERIES + 4,
            2,
            Rule::Chain(1, TS('i')),
            Rule::Chain(1, TS('l'))
        ),
        Rule(
            NS('K'), GRB_ERROR_SERIES + 8,
            2,

```

```

        Rule::Chain(3, TS('('), NS('W'), TS(')'),
        Rule::Chain(2, TS('('), TS(')))
    ),
    Rule(
        NS('N'), GRB_ERROR_SERIES + 6,
        14,
        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('c'), TS('('), NS('I'),
        TS(')'), TS('{'), NS('X'), TS(')'), NS('N')),
        Rule::Chain(4, TS('w'), NS('I'), TS(';'),
        NS('N')),
        Rule::Chain(4, TS('n'), NS('I'), TS(';'),
        NS('N')),

        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('c'), TS('('), NS('E'),
        TS(')'), TS('{'), NS('X'), TS(')'),
        Rule::Chain(7, TS('?'), TS('('), NS('R'),
        TS(')'), TS('{'), NS('X'), TS(')'),
        Rule::Chain(11, TS('?'), TS('('), NS('R'),
        TS(')'), TS('{'), NS('X'), TS(')'), TS('e'), TS('{'), NS('X'),
        TS('}')),
        Rule::Chain(3, TS('w'), NS('I'), TS(';')),
        Rule::Chain(3, TS('n'), NS('I'), TS(';'))
    ),

    Rule(
        NS('R'), GRB_ERROR_SERIES + 7,
        6,
        Rule::Chain(1, TS('i')),
        Rule::Chain(1, TS('l')),
        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::Chain(3, TS('l'), TS('v'), TS('l'))
    ),

    Rule(NS('E'), GRB_ERROR_SERIES + 4,

        8,
        Rule::Chain(1, TS('i')),
        Rule::Chain(1, TS('l')),
        Rule::Chain(2, TS('i'), NS('M')),

```

```

        Rule::Chain(2, TS('l'), NS('M')),
        Rule::Chain(3, TS('('), NS('E'), TS(')')),
        Rule::Chain(4, TS('('), NS('E'), TS(')')),
NS('M')),

        Rule::Chain(2, TS('i'), NS('K')),
        Rule::Chain(3, TS('i'), NS('K'), NS('M'))

    ),
    Rule(NS('M'), GRB_ERROR_SERIES + 4,
        1,
        Rule::Chain(2, TS('v'), NS('E'))
    ),
    Rule(NS('W'), GRB_ERROR_SERIES + 8,
        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('X'), GRB_ERROR_SERIES + 7,
        6,

        Rule::Chain(5, TS('i'), TS('='), NS('E'),
TS(';'), NS('X')),
        Rule::Chain(4, TS('w'), NS('I'), TS(';'),
NS('X')),
        Rule::Chain(4, TS('n'), NS('I'), TS(';'),
NS('X')),

        Rule::Chain(4, TS('i'), TS('='), NS('E'),
TS(';')),
        Rule::Chain(3, TS('w'), NS('I'), TS(';')),
        Rule::Chain(3, TS('n'), NS('I'), TS(';'))
    )
);
}

```


Приложение Г

Листинг 1 – Структура автомата магазинного типа

```

namespace MFST
{
#define MFST_TRACE_START
stream_out<<std::setw(4)<<std::left<<"Шар"<<": "\
<<std::setw(20)<<std::left<<"Правило"\
<<std::setw(30)<<std::left<<"Входная лента"\
<<std::setw(20)<<std::left<<"Стек"\
<<std::endl;
    int FST_TRACE_n = -1;
    char rbuf[205], sbuf[205], lbuf[1024];

#define NS(n) GRB::Rule::Chain::N(n);
#define TS(n) GRB::Rule::Chain::T(n);
#define ISNS(n) GRB::Rule::Chain::isN(n);

#define MFST_TRACE1
stream_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,lenta_position)\
<<std::setw(20)<<std::left<<getCSt(sbuf)\
<<std::endl;
#define MFST_TRACE2
stream_out<<std::setw(4)<<std::left<<FST_TRACE_n<<": "\
<<std::setw(20)<<std::left<<" "\
<<std::setw(30)<<std::left<<getCLenta(lbuf,lenta_position)\
<<std::setw(20)<<std::left<<getCSt(sbuf)\
<<std::endl;
#define MFST_TRACE3
stream_out<<std::setw(4)<<std::left<<++FST_TRACE_n<<": "\
<<std::setw(20)<<std::left<<" "\
<<std::setw(30)<<std::left<<getCLenta(lbuf,lenta_position)\
<<std::setw(20)<<std::left<<getCSt(sbuf)\
<<std::endl;
#define MFST_TRACE4(c)
stream_out<<std::setw(4)<<std::left<<++FST_TRACE_n<<":
"<<std::setw(20)<<std::left<<c<<std::endl;
#define MFST_TRACE5(c)
stream_out<<std::setw(4)<<std::left<<FST_TRACE_n<<":
"<<std::setw(20)<<std::left<<c<<std::endl;

```

```

#define MFST_TRACE6(c,k)
stream_out<<std::setw(4)<<std::left<<FST_TRACE_n<<":
"<<std::setw(20)<<std::left<<c<<k<<std::endl;
#define MFST_TRACE7
stream_out<<std::setw(4)<<std::left<<state.lenta_position<<":
"\

    <<std::setw(20)<<std::left<<rule.getCRule(rbuf,state.nrule
chain)\
                <<std::endl;

MfstState::MfstState()
{
    lenta_position = 0;
    nrule = -1;
    nrulechain = -1;
};
MfstState::MfstState(short pposition, MFSTSTSTACK pst,
short pnrulechain)
{
    lenta_position = pposition;
    st = pst;
    nrulechain = -pnrulechain;
};
MfstState::MfstState(short pposition, MFSTSTSTACK pst,
short pnrule, short pnrulechain)
{
    lenta_position = pposition;
    st = pst;
    nrule = pnrule;
    nrulechain = pnrulechain;
};
Mfst::MfstDiagnosis::MfstDiagnosis()
{
    lenta_position = -1;
    rc_step = SURPRISE;
    nrule = -1;
    nrule_chain = -1;
};
Mfst::MfstDiagnosis::MfstDiagnosis(short plenta_position,
RC_STEP prc_step, short pnrule, short pnrule_chain)
{
    lenta_position = plenta_position;
    rc_step = prc_step;
    nrule = pnrule;
    nrule_chain = pnrule_chain;
};
Mfst::Mfst() {
    lenta = 0;
    lenta_size = lenta_position = 0;
};
Mfst::Mfst(LT::LexTable plex, GRB::Greibach pgreibach)
{

```

```

        grebach = pgrebach;
        lex = plex;
        lenta = new short[lenta_size = lex.size];
        for (int k = 0; k < lenta_size; k++)
            lenta[k] = TS(lex.table[k].lexema);
        lenta_position = 0;
        st.push(grebach.stbottomT);
        st.push(grebach.startN);
        nrulechain = -1;
    };
Mfst::RC_STEP Mfst::step(std::ostream& stream_out)
{
    RC_STEP rc = SURPRISE;
    if (lenta_position < lenta_size)
    {
        if (GRB::Rule::Chain::isN(st.top()))
        {
            GRB::Rule rule;
            if ((nrule = grebach.getRule(st.top(), rule)) >=
0)
            {
                GRB::Rule::Chain chain;
                if ((nrulechain =
rule.getNextChain(lenta[lenta_position], chain, nrulechain +
1)) >= 0)
                {
                    MFST_TRACE1
                        savestate(stream_out); st.pop();
push_chain(chain); rc = NS_OK;
                    MFST_TRACE2
                }
                else
                {
                    MFST_TRACE4("TNS_NORULECHAIN/NS_NORULE")
                        savediagnosis(NS_NORULECHAIN); rc
= reststate(stream_out) ? NS_NORULECHAIN : NS_NORULE;
                };
            }
            else rc = NS_ERROR;
        }
        else if (st.top() == lenta[lenta_position])
        {
            lenta_position++; st.pop(); nrulechain = -1; rc
= TS_OK;
            MFST_TRACE3
        }
        else
        {
            MFST_TRACE4("TS_NOK/NS_NORULECHAIN")
                rc = reststate(stream_out) ? NS_NORULECHAIN
: NS_NORULE;
        };
    }
}

```

```

    }
    else
    {
        rc = LENTA_END;
        MFST_TRACE4("LENTA_END");
    };
    return rc;
};

bool Mfst::push_chain(GRB::Rule::Chain chain)
{
    for (int k = chain.size - 1; k >= 0; k--)
    st.push(chain.nt[k]);
    return true;
};

bool Mfst::savestate(std::ostream& stream_out)
{
    storestate.push(MfstState(lenta_position, st, nrule,
nrulechain));
    MFST_TRACE6("SAVESTATE:", storestate.size())
    return true;
};

```

Приложение Д

Начало трассировки синтаксического анализатора

Шаг	Правило	Входная лента	Стек
0---	S->tfiFBS	tfi(ti,ti){?(ivi){i=ivl;}	S\$
0---	SAVESTATE:	1	
0---		tfi(ti,ti){?(ivi){i=ivl;}	tfiFBS\$
1---		fi(ti,ti){?(ivi){i=ivl;}	fiFBS\$
2---		i(ti,ti){?(ivi){i=ivl;}	iFBS\$
3---		(ti,ti){?(ivi){i=ivl;}	(ti,ti)FBS\$
4---	F->(P)	(ti,ti){?(ivi){i=ivl;}	(ti,ti)FBS\$
4---	SAVESTATE:	2	
4---		(ti,ti){?(ivi){i=ivl;}	(ti,ti)(P)BS\$
5---		ti,ti){?(ivi){i=ivl;}	ti,ti)(P)BS\$
6---	P->ti	ti,ti){?(ivi){i=ivl;}	ti,ti)(P)BS\$
6---	SAVESTATE:	3	
6---		ti,ti){?(ivi){i=ivl;}	ti,ti)(ti)BS\$
7---		i,ti){?(ivi){i=ivl;}	i,ti)(ti)BS\$
8---		,ti){?(ivi){i=ivl;}	,ti)(ti)BS\$
9---	TS_NOK/NS_NORULECHAIN		
9---	RESTATE		
9---		ti,ti){?(ivi){i=ivl;}	ti,ti)(P)BS\$
10--	P->ti,P	ti,ti){?(ivi){i=ivl;}	ti,ti)(P)BS\$
10--	SAVESTATE:	3	
10--		ti,ti){?(ivi){i=ivl;}	ti,ti)(ti,P)BS\$
11--		i,ti){?(ivi){i=ivl;}	i,ti)(ti,P)BS\$
12--		,ti){?(ivi){i=ivl;}	,ti)(ti,P)BS\$
13--		ti){?(ivi){i=ivl;}	ti)(ti,P)BS\$
14--	P->ti	ti){?(ivi){i=ivl;}	ti)(ti,P)BS\$
14--	SAVESTATE:	4	

```

14--: -----ti){?(ivi){i=ivl;}e{i=(iv-----ti)BS$-
-----
15--: -----i){?(ivi){i=ivl;}e{i=(ivl-----i)BS$--
-----
16--: -----){?(ivi){i=ivl;}e{i=(ivl)-----)BS$---
-----
17--: -----{?(ivi){i=ivl;}e{i=(ivl)v-----BS$----
-----
18--: B->{NrI;}-----{?(ivi){i=ivl;}e{i=(ivl)v-----BS$----
-----
18--: SAVESTATE:-----5

```

Конец трассировки синтаксического анализатора

```

1057: RESTATE-----
1057: -----dti=l;wi;}-----N}$----
-----
1058: N->dti=E;N-----dti=l;wi;}-----N}$----
-----
1058: SAVESTATE:-----75
1058: -----dti=l;wi;}-----
dti=E;N}$-----
1059: -----ti=l;wi;}-----
ti=E;N}$-----
1060: -----i=l;wi;}-----
i=E;N}$-----
1061: -----=l;wi;}-----=E;N}$-
-----
1062: -----l;wi;}-----E;N}$--
-----
1063: E->l-----l;wi;}-----E;N}$--
-----
1063: SAVESTATE:-----76
1063: -----l;wi;}-----l;N}$--
-----
1064: -----;wi;}-----;N}$---
-----
1065: -----wi;}-----N}$----
-----
1066: N->wI;N-----wi;}-----N}$----
-----
1066: SAVESTATE:-----77
1066: -----wi;}-----wI;N}$-
-----
1067: -----i;}-----I;N}$--
-----
1068: I->i-----i;}-----I;N}$--
-----
1068: SAVESTATE:-----78
1068: -----i;}-----i;N}$--
-----
1069: -----; }-----;N}$---
-----

```

```

1070:  -----}-----N}$---
-----
1071: TNS_NORULECHAIN/NS_NORULE
1071: RESTATE-----
1071:  -----i;}-----I;N}$--
-----
1072: TNS_NORULECHAIN/NS_NORULE
1072: RESTATE-----
1072:  -----wi;}-----N}$---
-----
1073: N->wI;-----wi;}-----N}$---
-----
1073: SAVESTATE:-----77
1073:  -----wi;}-----wI;}$--
-----
1074:  -----i;}-----I;}$---
-----
1075: I->i-----i;}-----I;}$---
-----
1075: SAVESTATE:-----78
1075:  -----i;}-----i;}$---
-----
1076:  -----;}-----;}$---
-----
1077:  -----}-----}$-----
-----
1078:  -----}$-----
-----
1079: LENTA_END-----
1080: ----->LENTA_END-----

```

Приложение Е

Листинг 1 – Код на языке Assembler

```
.586
.model flat, stdcall
includelib libcrt.lib
includelib kernel32.lib
includelib ../Debug/StaticLib.lib
ExitProcess PROTO :DWORD

MathPow PROTO :DWORD, :DWORD
MathRand PROTO :DWORD, :DWORD
OutputStr PROTO :DWORD
OutputStrNoLine PROTO :DWORD
OutputInt PROTO :DWORD
OutputIntNoLine PROTO :DWORD

.stack 4096

.CONST
    overflownum BYTE "Выход за пределы значения", 0
    neguint BYTE "Отрицательное число", 0
    L1 DWORD 4
    L2 DWORD 3
    L3 BYTE "x > y", 0
    L4 BYTE "x < y", 0
    L5 DWORD 0
    L6 DWORD 21
    L7 DWORD 177
    L8 DWORD 346
    L9 BYTE "numa: ", 0
    L10 BYTE "numb: ", 0
    L11 DWORD 15
    L12 DWORD 2
    L13 DWORD 22136
    L14 DWORD 13
    L15 BYTE "hello world!!!", 0

.data
    count DWORD 0
    main_numa DWORD 0
    main_numb DWORD 0
    main_res DWORD 0
    main_i DWORD 0
    main_y DWORD 0
    main_text DWORD ?

.code

change PROC change_a : DWORD, change_b : DWORD
    mov eax, change_a
    cmp eax, change_b
```



```

        je m0
        jne m1
        je m1
m0:
        push change_b
        push L1
        pop ecx
CYCLE0:
        pop eax
        shl eax, 1
        cmp eax, 4294967295
ja OVERFLOW
        cmp eax, 0
jl NEGNUM
        push eax
        loop CYCLE0
        pop change_a
        jmp e0
m1:
        push change_a
        push L2
        pop ecx
CYCLE1:
        pop eax
        shl eax, 1
        cmp eax, 4294967295
ja OVERFLOW
        cmp eax, 0
jl NEGNUM
        push eax
        loop CYCLE1
        push change_b
        pop eax
        pop ebx
        add eax, ebx
        cmp eax, 4294967295
ja OVERFLOW
        cmp eax, 0
jl NEGNUM
        push eax
        pop change_a
e0:
        push change_a
        jmp local0
local0:
        pop eax
        ret
OutAsm:
        push 0
        call ExitProcess
OVERFLOW:
        push offset overflownum
        call OutputStr

```

```

        push 0
        call ExitProcess
NEGNUM:
        push offset neguint
        call OutputStr
        push 0
        call ExitProcess
change ENDP

checknums PROC checknums_x : DWORD, checknums_y : DWORD
        mov eax, checknums_x
        cmp eax, checknums_y
        ja m2
        jb m3
        je m3
m2:
        push offset L3
        call OutputStr
        jmp e1
m3:
        push offset L4
        call OutputStr
e1:
        push L5
        jmp local1
local1:
        pop eax
        ret
OutAsm:
        push 0
        call ExitProcess
OVERFLOW:
        push offset overflownum
        call OutputStr
        push 0
        call ExitProcess
NEGNUM:
        push offset neguint
        call OutputStr
        push 0
        call ExitProcess
checknums ENDP

main PROC
        push L6
        push L1
        pop edx
        pop edx
        push L1
        push L6
        call MathPow
        push eax
        pop main_numa

```

```

    push L7
    push L8
    pop edx
    pop edx
    push L8
    push L7
    call MathRand
    push eax
    pop main_numb
    push offset L9
    call OutputStrNoLine
    push main_numa
    call OutputInt
    push offset L10
    call OutputStrNoLine
    push main_numb
    call OutputInt
    push main_numa
    push main_numb
    pop edx
    pop edx
    push main_numb
    push main_numa
    call checknums
    push eax
    pop main_res
    push main_res
    call OutputInt
    push L11
    pop main_i
    push main_i
    push L12
    pop ecx
CYCLE2:
    pop eax
    shr eax, 1
    cmp eax, 4294967295
ja OVERFLOW
    cmp eax, 0
jl NEGNUM
    push eax
    loop CYCLE2
    push L13
    push L14
    pop edx
    pop edx
    push L14
    push L13
    call change
    push eax
    pop eax
    pop ebx
    add eax, ebx

```

```

        cmp eax, 4294967295
ja OVERFLOW
        cmp eax, 0
jl NEGNUM
        push eax
        push main_numb
        pop ebx
        pop eax
        sub eax, ebx
        push eax
        cmp eax, 4294967295
ja OVERFLOW
        cmp eax, 0
jl NEGNUM
        pop main_y
        push main_y
        call OutputInt
        push offset L15
        pop main_text
        push main_text
        call OutputStrNoLine
OutAsm:
        push 0
        call ExitProcess
OVERFLOW:
        push offset overflownum
        call OutputStr
        push 0
        call ExitProcess
NEGNUM:
        push offset neguint
        call OutputStr
        push 0
        call ExitProcess
main ENDP

end main

```