МИНИСТЕРСТВО ОБРАЗОВАНИЯ РЕСПУБЛИКИ БЕЛАРУСЬ

Учреждение образования «БЕЛОРУССКИЙ ГОСУДАРСТВЕННЫЙ ТЕХНОЛОГИЧЕСКИЙ УНИВЕРСИТЕТ»

Факультет Информационных Технологий
Кафедра Программной инженерии
Специальность ПОИТ

ПОЯСНИТЕЛЬНАЯ ЗАПИСКА
К КУРСОВОМУ ПРОЕКТУ НА ТЕМУ:

«Разработка компилятора KVV-2023»

Выполнил студент Корнелюк Валентин Владимирович
(Ф.И.О.)

Руководитель проекта доц. Наркевич Аделина Сергеевна
(учен. степень, звание, должность, подпись, Ф.И.О.)

Заведующий кафедрой доц. Смелов Владимир Владиславович
(учен. степень, звание, должность, подпись, Ф.И.О.)

Консультанты доц. Наркевич Аделина Сергеевна
(учен. степень, звание, должность, подпись, Ф.И.О.)

Курсовой проект защищен с оценкой

Минск 2023

Оглавление

Введение 4
1 Спецификация языка программирования 5
1.1 Характеристика языка программирования 5
1.2 Алфавит языка 5
1.3 Применяемые сепараторы 5
1.4 Применяемые кодировки 6
1.5 Типы данных 6
1.6 Преобразование типов данных 7
1.7 Идентификаторы 7
1.8 Литералы 7
1.9 Объявление данных 8
1.10 Инициализация данных 8
1.11 Инструкции языка 9
1.12 Операции языка 9
1.13 Выражения и их вычисление 10
1.14 Конструкции языка 10
1.15 Области видимости идентификаторов 11
1.16 Семантические проверки 11
1.17 Распределение оперативной памяти на этапе выполнения 12
1.18 Стандартная библиотека и её состав 12
1.19 Ввод и вывод данных 12
1.20 Точка входа 13
1.21 Препроцессор 13
1.22 Соглашения о вызовах 13
1.23 Объектный код 13
1.24 Классификация сообщений транслятора 13
1.25 Контрольный пример 13
2 Структура транслятора 15
2.1 Компоненты транслятора, их назначение и принципы взаимодействия 15
2.2 Перечень параметров транслятора 16
2.3 Протоколы, формируемые транслятором 17
3 Разработка лексического анализатора 18
3.1 Структура лексического анализатора 18
3.2. Входные и выходные данные лексического анализатора 19
3.3. Параметры лексического анализатора 19
3.4 Алгоритм лексического анализа 19
3.5. Контроль входных символов 20
3.6 Удаление избыточных символов 20
3.7. Перечень ключевых слов 21
3.8. Основные структуры данных 23
3.9. Структура и перечень сообщений лексического анализатора 25
3.10. Принцип обработки ошибок 25

3.10 Контрольный пример ... 26
4. Разработка синтаксического анализатора ... 27
4.1 Структура синтаксического анализатора ... 27
4.2 Контекстно-свободная грамматика, описывающая синтаксис языка ... 27
4.3 Построение конечного магазинного автомата ... 29
4.4 Основные структуры данных ... 30
4.5 Описание алгоритма синтаксического разбора ... 31
4.6. Параметры синтаксического анализатора и режимы его работы ... 31
4.7. Структура и перечень сообщений синтаксического анализатора ... 31
4.8. Принцип обработки ошибок ... 32
4.9. Контрольный пример ... 32
5 Разработка семантического анализатора ... 33
5.1 Структура семантического анализатора ... 33
5.2 Функции семантического анализатора ... 33
5.3 Структура и перечень сообщений семантического анализатора ... 33
5.4 Принцип обработки ошибок ... 34
5.5 Контрольный пример ... 34
6. Вычисление выражений ... 35
6.1 Выражения, допускаемые языком ... 35
6.2 Польская запись и принцип её построения ... 35
6.3 Программная реализация обработки выражений ... 36
6.4 Контрольный пример ... 36
7. Генерация кода ... 37
7.1 Структура генератора кода ... 37
7.2 Представление типов данных в оперативной памяти ... 37
7.3 Статическая библиотека ... 38
7.4 Особенности алгоритма генерации кода ... 38
7.5 Параметры, управляющие генерацией кода ... 39
7.6 Контрольный пример ... 40
8. Тестирование транслятора ... 41
8.1 Тестирование проверки на допустимость символов ... 41
8.2 Тестирование лексического анализатора ... 41
8.3 Тестирование синтаксического анализатора ... 41
8.4 Тестирование семантического анализатора ... 43
Заключение ... 45
Список использованных источников ... 46
Приложение А ... 47
Приложение Б ... 2
Приложение В ... 46
Приложение Г ... 51
Приложение Д ... 55

Введение

В данном курсовом проекте поставлена задача разработки собственного языка программирования и транслятора для него. Название языка – KVV-2023.
Этот язык программирования предназначен для выполнения простейших операций и арифметических действий над числами. Написание транслятора будет осуществляться на языке C++.

Исходя из цели курсового проекта, были определены следующие задачи:

- Разработка спецификации языка KVV-2023;
- Разработка лексического анализатора;
- Разработка синтаксического анализатора;
- Разработка семантического анализатора;
- Разбор арифметических выражений;
- Разработка генератора кода;
- Тестирование транслятора.

Решения каждой и поставленных задач будут приведены в соответствующих главах курсового проекта.

1 Спецификация языка программирования

1.1 Характеристика языка программирования

Язык программирования KVV-2023 является процедурным, универсальным строго типизированным, компилируемым.

1.2 Алфавит языка

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

Алфавит языка KVV-2023 состоит из следующих множеств символов:

– латинские символы верхнего и нижнего регистра: {A, B, C, …, Z, a, b, c, …, z};

– цифры: {0, 1, 2, 3, 4, 5, 6, 7, 8, 9};

– символы пробела, табуляции и перевода строки;

– знаки пунктуации языка: {(), {}, [], ;, :, #, +, -, /, *, =, <>, &,!}.

1.3 Применяемые сепараторы

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

Таблица 1.1 – Символы-сепараторы

Сепаратор	Назначение
=	Оператор присваивания
‘пробел’	Разделитель цепочек. Допускается везде кроме названий идентификаторов и ключевых слов
[…]	Блок функции или условной конструкции/цикла
(…)	Блок фактических или формальных параметров функции, а также приоритет арифметических операций
,	Разделитель параметров функций
#	Символ, отделяющий условные конструкции/циклы
+ - */	Арифметические операции
> < & !	Логические операции (операции сравнения: больше, меньше, проверка на равенство, на неравенство), используемые в условии цикла/условной конструкции.
;	Разделитель программных конструкций
}{	Операторы сдвигов

Использование сепараторов важно для правильного синтаксического анализа и интерпретации кода, а также для обеспечения корректной работы различных программных конструкций.

1.4 Применяемые кодировки

Для написания программ язык KVV-2023 использует кодировку ASCII, содержащую английский алфавит, а также некоторые специальные символы, такие как [ ] ( ) , ; : # + - / * > < & !{}.

	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
00	00 □	01 □	02 □	03 □	04 □	05 □	06 □	07 □	08 □	09 □	0A □	0B □	0C □	0D □	0E □	0F □
10	10 □	11 □	12 □	13 □	14 □	15 □	16 □	17 □	18 □	19 □	1A □	1B □	1C □	1D □	1E □	1F □
20	20	21 !	22 “	23 #	24 $	25 %	26 &	27 (	28)	29 '	2A *	2B +	2C ,	2D -	2E .	2F /
30	30 0	31 1	32 2	33 3	34 4	35 5	36 6	37 7	38 8	39 9	3A :	3B ;	3C <	3D =	3E >	3F ?
40	40 @	41 A	42 B	43 C	44 D	45 E	46 F	47 G	48 H	49 I	4A J	4B K	4C L	4D M	4E N	4F O
50	50 P	51 Q	52 R	53 S	54 T	55 U	56 V	57 W	58 X	59 Y	5A Z	5B [	5C \	5D]	5E ^	5F _
60	60 `	61 a	62 b	63 c	64 d	65 e	66 f	67 g	68 h	69 i	6A j	6B k	6C l	6D m	6E o	6F p
70	70 p	71 q	72 r	73 s	74 t	75 u	76 v	77 w	78 x	79 y	7A z	7B {	7C |	7D }	7E ~	7F □

Рисунок 1.1 Алфавит входных символов

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

1.5 Типы данных

В языке KVV-2023 реализованы два фундаментальных типа данных: целочисленный и строковый. Описание типов приведено в таблице 1.2.

Таблица 1.2 – Типы данных языка KVV-2023

Тип данных	Характеристика
Целочисленный тип данных **number**	Фундаментальный тип данных. Используется для работы с числовыми значениями. В памяти занимает 1 байт. Максимальное значение: 127. Минимальное значение: -128. Инициализация по умолчанию: значение 0. Поддерживаемые операции: **+** (бинарный) – оператор сложения; **-** (бинарный) – оператор вычитания; ***** (бинарный) – оператор умножения; **/** (бинарный) – оператор деления;

Окончание таблицы 1.2

Тип данных	Характеристика
	= (бинарный) – оператор присваивания; В качестве условия цикла/условного оператора поддерживаются следующие логические операции: > (бинарный) – оператор «больше»; < (бинарный) – оператор «меньше»; **!** (бинарный) – оператор проверки на неравенство; { (бинарный) – оператор сдвига влево; } (бинарный) – оператор сдвига вправо.
Строковый тип данных **string**	Фундаментальный тип данных. Используется для работы с символами, каждый из которых занимает 1 байт. Максимальное количество символов – 255. Инициализация по умолчанию: строка нулевой длины “”. Операции над данными строкового типа: присваивание строковому идентификатору значения другого строкового идентификатора, строкового литерала или значения строковой функции, а также использование библиотечных функций.

Эти типы данных обеспечивают разнообразные возможности для представления и манипуляции данными в программе.

1.6 Преобразование типов данных

В языке программирования KVV-2023 присутствует преобразование строки в число с помощью функции стандартной библиотеки atoii(string), которая возвращает значение типа number.

1.7 Идентификаторы

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

1.8 Литералы

С помощью литералов осуществляется инициализация переменных. Все

литералы являются **rvalue**. Литералы rvalue - это значения, которые обычно не имеют имени или явного места в памяти. "Rvalue" означает "значение, которое может быть использовано справа от оператора присваивания". В контексте литералов это обычно относится к константам или временным значениям, которые создаются на месте использования и не сохраняются в явном виде. Типы литералов языка KVV-2023 представлены в таблице 1.3.

Таблица 1.3 – Литералы

Литералы	Пояснение
Целочисленные литералы в десятичном представлении	Последовательность цифр 0…9 с предшествующим знаком минус или без него (знак минус не должен отделяться пробелом).
Целочисленные литералы в восьмеричном представлении	Символ 0(положительный) либо 1 (отрицательный), задающие знак литерала, символ ‘q’, указывающий на восьмеричный литерал, последовательность цифр 0…7 .
Строковые литералы	Набор символов (от 1 до 255 символов), заключённых в двойные кавычки.

Ограничения на строковые литералы языка KVV-2023: внутри литерала не допускается использование одинарных и двойных кавычек. Ограничения на целочисленные литералы: не могут начинаться с нуля, если их значение не ноль; если литерал отрицательный, после знака “-” не может быть нуль.

1.9 Объявление данных

Для объявления переменной используется ключевое слово **new**, после которого указывается тип данных и имя идентификатора. Допускается инициализация при объявлении. Конструкция для объявления переменных:

new <тип данных> <идентификатор>;

Для объявления функций используется ключевое слово **function**, перед которым указывается тип функции (если функция возвращает значение), или ключевое слово **procedure**, если функция ничего не возвращает, а после – имя функции либо процедуры. Далее обязателен список параметров и тело функции.

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 только идентификаторы или литералы. При объявлении без инициализации предусмотрены значения по умолчанию: значение 0 для типа **number** и строка длины 0 (“”) для типа **string**.

Пример объявления числового типа с инициализацией:
new number num1 = -1;
new number num2 = q80;
Пример объявления переменной строкового типа с инициализацией:
new string str1= “hello world”;

1.11 Инструкции языка

Инструкции языка KVV-2023 представлены в таблице 1.4.

Таблица 1.4 – Инструкции языка KVV-2023

Инструкция	Реализация
Объявление переменной	**new** <тип данных> <идентификатор>;
Объявление переменной с явной инициализацией	**new** <тип данных> <идентификатор> = <значение>; Значение – инициализатор конкретного типа. Может быть только литералом или идентификатором.
Возврат из функции или процедуры(после вызова)	Для функций, возвращающих значение: **return** <идентификатор/литерал>; Для процедур: **return;**
Вывод данных	**write** <идентификатор/литерал>;
Вызов функции или процедуры	<идентификатор функции> (<список параметров>); Список параметров может быть пустым.
Перевод строки	**newline**;
Присваивание	<идентификатор> = <выражение>; Выражением может быть идентификатор, литерал, или вызов функции соответствующего типа. Для целочисленного типа выражение может быть дополнено арифметическими операциями с любым количеством операндов с использованием скобок. Для строкового типа выражение может быть только идентификатором, литералом или вызовом функции, возвращающей значение строкового типа.

Инструкции представляют собой базовые операции, которые выполняются пошагово в процессе выполнения программы. Каждая инструкция выполняет конкретное действие, и они объединяются вместе для создания функциональных программ. Инструкции представляют собой "строительные блоки" программы, определяя её поведение при выполнении.

1.12 Операции языка

В языке KVV-2023 предусмотрены операции, представленные в таблице 1.5.

Таблица 1.5 – Операции языка KVV-2023

Тип оператора	Оператор
Арифметические	+ – сложение - – вычитание * – умножение / – деление = – присваивание
Строковые	= – присваивание
Логические	> – больше < – меньше ! – проверка на неравенство &- проверка на равенство
Сдвиговые	} – сдвиг вправо { – сдвиг влево

Приоритетность операции умножения выше приоритета операций сложения и вычитания. Для установки наивысшего приоритета используются круглые скобки.

1.13 Выражения и их вычисление

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

- Допускается использовать скобки для смены приоритета операций;
- Выражение записывается в строку без переносов;
- Использование двух подряд идущих операторов не допускается;
- Допускается использовать в выражении вызов функции, вычисляющей и возвращающей целочисленное значение.

1.14 Конструкции языка

Программа на языке KVV-2023 оформляется в виде функций пользователя и главной функции. При составлении функций рекомендуется выделять блоки и фрагменты и применять отступы для лучшей читаемости кода.

Программные конструкции языка представлены в таблице 1.6.

Таблица 1.6 – Программные конструкции языка KVV-2023

Конструкция	Реализация
Главная функция	**main** **[** … **]**
Внешняя функция	<тип данных> **function** <идентификатор> **(**<тип> <идентификатор>, ...**)** **[**…

Окончание таблицы 1.6

Конструкция	Реализация
	return <идентификатор/литерал>; **]**
Внешняя процедура	**procedure function** <идентификатор> **(**<тип> <идентификатор>, ...**)** **[**… **return**; **]**
Цикл	**condition**: <идентификатор1> <оператор> <идентификатор2> # **cycle** [… **]** #
Условная конструкция	**condition:** <идентификатор1> <оператор> <идентификатор2> # **istrue [**…**]** **isfalse [**…**]** #

Цикл (операторы внутри блока **cycle**) выполняется, пока истинно условие <идентификатор1> <оператор> <идентификатор2>.

Идентификаторы в условной конструкции <идентификатор1>, <идентификатор2> - это идентификаторы или литералы целочисленного типа. <оператор> - один из операторов сравнения (**> < & !**), устанавливающий отношение между двумя операндами и организующий условие данной конструкции. При истинности условия выполняется код внутри блока **istrue**, иначе – код внутри блока **isfalse**. Любой из блоков **istrue**, **isfalse** может отсутствовать, но не оба блока одновременно. При отсутствии одного из блоков, в зависимости от истинности или ложности условия программа может как выполнить один из заявленных блоков, так и передать управление инструкции, следующей в коде за закрывающим условную конструкцию символом **‘#’**.

1.15 Области видимости идентификаторов

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

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

1.16 Семантические проверки

В языке программирования KVV-2023 выполняются следующие семантические проверки:

- Наличие функции **main** – точки входа в программу;
- Единственность точки входа;
- Переопределение идентификаторов;

– Использование идентификаторов без их объявления;
– Проверка соответствия типа функции и возвращаемого параметра;
– Правильность передаваемых в функцию параметров: количество, типы;
– Правильность строковых выражений;
– Превышение размера строковых и числовых литералов;
– Правильность составленного условия цикла/условного оператора.

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.

1.18 Стандартная библиотека и её состав

В языке KVV-2023 присутствует стандартная библиотека, которая подключается автоматически при трансляции исходного кода в язык ассемблера. Содержимое библиотеки и описание функций представлено в таблице 1.8.

Таблица 1.8 - Стандартная библиотека языка KVV-2023

Функция	Описание
string concat(**string** s1, **string** s2);	Строковая функция, выполняет объединение строк s1 и s2 в указанном порядке. Возвращает строку – результат объединения строк s1 и s2.
number atoii(**string** str);	Целочисленная функция. Преобразует строку в число.
number lenght(**string** str);	Целочисленная функция. Вычисляет и возвращает длину строки str.
void outnum(int value)	Функция для вывода в стандартный поток значения целочисленного идентификатора/литерала.
void outstr(char* line)	Функция для вывода в стандартный поток значения строкового идентификатора/литерала.

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

1.19 Ввод и вывод данных

Вывод данных осуществляется с помощью оператора **write**. Допускается использование оператора **write** с литералами и идентификаторами.

Функции, управляющие выводом данных, реализованы на языке C++ и

вызываются из транслированного кода, конечному пользователю недоступны. Пользовательская команда **write** в транслированном коде будут заменена вызовом нужных библиотечных функций. Библиотека, содержащая нужные процедуры, подключается на этапе генерации кода.

Ввод данных не предусмотрен.

1.20 Точка входа

В языке KVV-2023 каждая программа должна содержать главную функцию (точку входа) **main**, с первой инструкции которой начнётся последовательное выполнение команд программы.

1.21 Препроцессор

Препроцессор, принимающий и выдающий некоторые данные на вход транслятору, в языке KVV-2023 отсутствует.

1.22 Соглашения о вызовах

В языке KVV-2023 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 KVV-2023 транслируется в язык ассемблера, а затем - в объектный код.

1.24 Классификация сообщений транслятора

Генерируемые транслятором сообщения определяют степень его информативности, то есть сообщения транслятора должны давать максимально полную информацию о допущенной пользователем ошибке при написании программы. Сообщения транслятора приведены в таблице 1.10.

Таблица 1.10 Классификация ошибок

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

Эта классификация является важным элементом в обеспечении

информативности сообщений транслятора. Номера ошибок охватывают различные аспекты трансляции, начиная от системных ошибок и заканчивая ошибками в лексическом, синтаксическом и семантическом анализе.

1.25 Контрольный пример

Контрольный пример демонстрирует главные особенности языка KVV-2023: его фундаментальные типы, основные структуры, функции, процедуры, использование функций статической библиотеки. Исходный код контрольного примера представлен в приложении А.

2 Структура транслятора

2.1 Компоненты транслятора, их назначение и принципы взаимодействия

В языке KVV-2023 исходный код транслируется в язык Assembler. Транслятор языка разделён на отдельные части, которые взаимодействуют между собой и выполняют отведённые им функции, которые представлены в пункте 2.1. Для того чтобы получить ассемблерный код, 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 KVV-2023 приведена на рисунке 2.1.

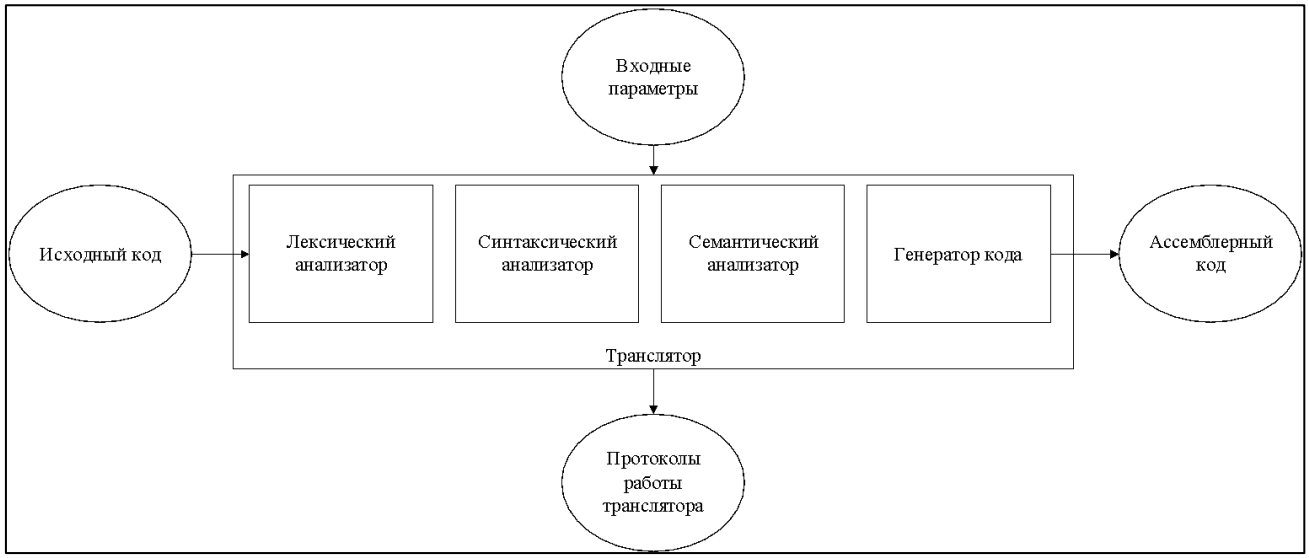

Рисунок 2.1 Структура транслятора языка программирования KVV-2023

Первая стадия работы компилятора называется лексическим анализом, а программа, её реализующая – лексическим анализатором. На вход лексического анализатора подаётся последовательность символов входного языка. Он производит предварительный разбор текста, преобразующий единый массив текстовых символов в массив отдельных слов (в теории компиляции вместо термина «слово» часто используют термин «токен»). Примеры лексических единиц: идентификаторы, числа, символы операций, служебные слова и т.д. Лексический анализатор преобразует исходный текст, заменяя лексические единицы их внутренним представлением – лексемами, для создания промежуточного представления исходной программы. Каждой лексеме сопоставляется ее тип и запись в таблице идентификаторов, в которой хранится дополнительная информация. Таблица лексем (ТЛ) и таблица идентификаторов (ТИ) являются входом для следующей фазы компилятора – синтаксического анализа (разбора).

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 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 KVV-2023

Входной параметр	Описание параметра	Значение по умолчанию
-in:<путь к in-файлу>	Файл с исходным кодом на языке KVV-2023, имеющий расширение .txt. Параметр является обязательным.	Не предусмотрено
-log:<путь к log-файлу>	Файл журнала для вывода протоколов работы программы. Параметр не является обязательным.	Значение по умолчанию: <имя in-файла>.log
-tb	Управляет выводом в консоль таблиц лексем и идентификаторов.	Если параметр не задан, то таблицы выводится не будут.

Входные параметры транслятора представляют собой опции, передаваемые при компиляции программы, чтобы настроить различные аспекты компиляции. Они предоставляют программистам возможность управлять компиляцией, оптимизировать процесс и обеспечивать удобный механизм отслеживания работы транслятора.

2.3 Протоколы, формируемые транслятором

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

Таблица 2.2 – Протоколы, формируемые транслятором языка KVV-2023

Формируемый протокол	Описание выходного протокола
Файл журнала, заданный параметром "-log:"	Файл с протоколом работы транслятора языка программирования KVV-2023. Содержит таблицу лексем и таблицу идентификаторов, протокол работы синтаксического анализатора и дерево разбора, полученные на этапе лексического и синтаксического анализа, а также результат работы алгоритма преобразования выражений к польской записи.
Дерево разбора	Результат работы синтаксического анализатора.
Выходной файл, с расширением ".asm"	Результат работы программы – файл, содержащий исходный код на языке ассемблера.

Важным этапом работы транслятора языка KVV-2023 является формирование протоколов, которые описывают процесс работы лексического, синтаксического и семантического анализаторов. Такие протоколы и файлы обеспечивают полную документацию о процессе трансляции, позволяя анализировать и проверять работу компилятора KVV-2023.

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 простейшие конструкции языка. Лексический анализатор производит предварительный разбор текста, преобразующий единый массив текстовых символов в массив токенов.

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

Функции лексического анализатора:

− удаление «пустых» символов.
− распознавание идентификаторов и ключевых слов;
− распознавание литералов;
− распознавание разделителей и знаков операций.

Структура лексического анализатора представлена на рисунке 3.1.

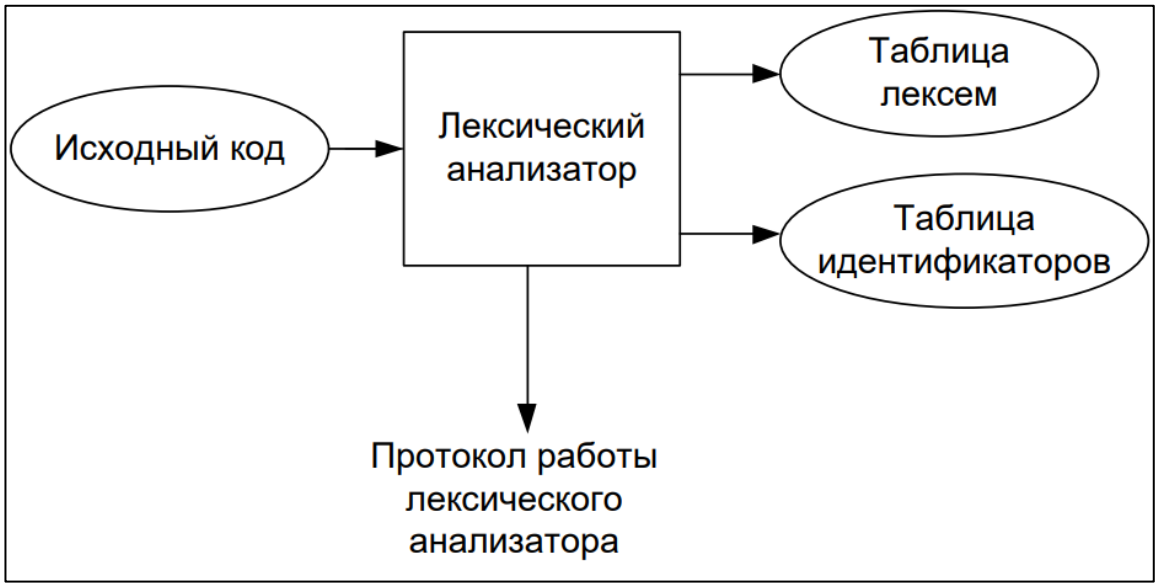

Рисунок 3.1 Структура лексического анализатора

Лексический анализатор транслятора языка KVV-2023 выделяет лексемы из исходного кода и формирует таблицы лексем и идентификаторов. Этот этап представляет собой первичный шаг в преобразовании исходного кода, создавая основу для следующих этапов синтаксического и семантического анализа.

3.2. Входные и выходные данные лексического анализатора

На вход лексического анализатора подаётся последовательность символов входного языка. 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. Выходными данными являются таблица лексем и таблица идентификаторов.

3.3. Параметры лексического анализатора

Для формирования файлов с результатами работы лексического анализатора используются входные параметры, которые приведены в таблице 2.1.

3.4 Алгоритм лексического анализа

Алгоритм лексического анализа:

- проверяет входной поток символов программы на исходном языке на допустимость, удаляет лишние пробелы и добавляет сепаратор для вычисления номера строки для каждой лексемы;
- для выделенной части входного потока выполняется функция распознавания лексемы;
- при успешном распознавании информация о выделенной лексеме заносится в таблицу лексем и таблицу идентификаторов, и алгоритм возвращается к первому этапу;
- формирует протокол работы;
- при неуспешном распознавании выдается сообщение об ошибке.

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

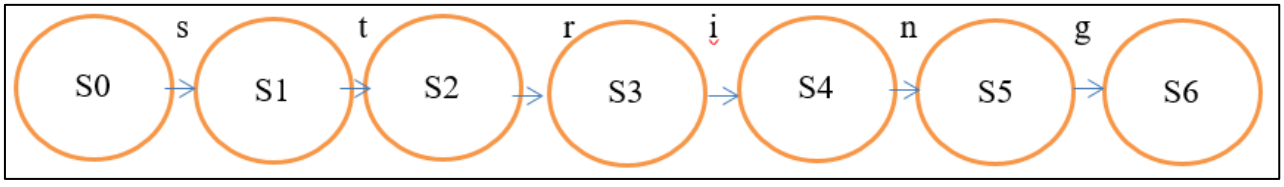

Рисунок 3.2 Пример графа переходов для цепочки string

Алгоритм лексического анализа представляет собой четко структурированный процесс обработки исходного кода программы. Этот алгоритм обладает высокой гибкостью и точностью, позволяя эффективно обрабатывать разнообразные выражения на исходном языке.

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   {\
        IN::F, IN::F, IN::F, IN::F,  IN::F, IN::F, IN::F, IN::F,   IN::F, IN::P, IN::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, IN::F,  IN::F, IN::F, IN::F, IN::F,\
        IN::P, IN::S, IN::Q, IN::S,  IN::T, IN::T, IN::S, IN::F,   IN::S, IN::S, IN::S, IN::S,  IN::S, IN::S, IN::T, IN::S,\
        IN::T, IN::T, IN::T, IN::T,  IN::T, IN::T, IN::T, IN::T,   IN::T, IN::T, IN::T, IN::S,  IN::S, IN::S, IN::S, 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, 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::S,  IN::T, IN::S, IN::T, IN::T,\
        IN::F, 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, IN::T, IN::T, IN::T,  IN::T, IN::T, IN::T, IN::T,   IN::T, IN::T, IN::T, IN::S,  IN::T, IN::T, IN::S, IN::T,\
                                                                                                                        \
        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,\
        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,\
        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,\
        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,\
        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,\
        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,\
        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,\
        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 \
}
```

Рисунок 3.2. Таблица контроля входных символов

Таблица 3.1 Соответствие символов и их значений в таблице

Значение в таблице входных символов	Символы
Разрешенный	T
Запрещенный	F
Игнорируемый	I
Литерал	Q
Сепаратор	S
Перевод строки	N
Пробел, табуляция	P

Контроль входных символов в лексическом анализаторе обеспечивает структурирование исходного кода путем разделения символов по категориям. Такой подход позволяет создать удобные условия для последующей обработки, анализа и интерпретации кода.

3.6 Удаление избыточных символов

Избыточными символами являются символы табуляции и пробелы.

Избыточные символы удаляются на этапе разбиения исходного кода на токены.

Описание алгоритма удаления избыточных символов:

- Посимвольно считываем файл с исходным кодом программы;
- Встреча пробела или знака табуляции является своего рода встречей

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

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
number, string	t	Названия типов данных языка.
Идентификатор	i	Длина идентификатора – 8 символов.
Литерал	l	Литерал любого доступного типа.
Восьмиричный литерал	q	Литерал в восьмиричном представлении.
function	f	Объявление функции.
procedure	p	Ключевое слово для процедур – функций, не возвращающих значения. Указывается перед словом function.
return	e	Выход из функции/процедуры.
main	m	Главная функция.
new	n	Объявление переменной.
write	@	Вывод данных.
condition:	?	Указывает начало цикла/условного оператора.
istrue	r	Истинная ветвь условного оператора.
isfalse	w	Ложная ветвь условного оператора.
cycle	c	Указывает на начало тела цикла.
newline	^	Оператор вывода символа перевода строки.
#	#	Разделение конструкций в цикле/условном операторе.
;	;	Разделение выражений.
,	,	Разделение параметров функций.
[	[	Начало блока/тела функции.
]	]	Закрытие блока/тела функции.
} {	} {	Знаки сдвиговых операций.
(	(	Передача параметров в функцию, приоритет операций.
)	)	Закрытие блока для передачи параметров, приоритет операций.
=	=	Знак присваивания.

Окончание таблицы 3.2

Токен	Лексема	Пояснение
+ - * /	+ - * /	Знаки арифметических операций.
>< < ! ?	>< < ! ?	Знаки логических операторов

Пример реализации таблицы лексем представлен в приложении Б.

Каждому выражению соответствует детерминированный конечный автомат, по которому происходит разбор данного выражения. На каждый автомат в массиве подаётся токен и с помощью регулярного выражения, соответствующего данному графу переходов, происходит разбор. В случае успешного разбора выражения оно записывается в таблицу лексем. Если выражение является идентификатором или литералом, информация также заносится в таблицу идентификаторов. Структура конечного автомата и пример графа перехода конечного автомата изображены в листингах 3.1 и 3.2 соответственно.

```
struct RELATION          // ребро: символ -> вершина графов переходов
КА
     {
          char  symbol;  // символ перехода
          short nnode;   // номер смежной вершины
          RELATION(
               char c,        // символ перехода
               short ns  // новое состояние
          );
     };

     struct NODE                         //вершина графа переходов
     {
          short n_relation;         //количество инциндентных ребер
          RELATION* relations;      //инциндентные ребра
          NODE();                         //конструктор без параметров
          NODE(short n, RELATION rel, ...);  //количество
инциндентных ребер, список ребер
     };

     struct FST   //недетерминированный конечный автомат
     {
          char* string;                  //цепочка(строка, завершается
0x00)
          short position;                     //текущая позиция в
цепочке
          short nstates;                 //количество состояний
автомата
```

```
        NODE* node;                          //граф переходов:[0]-
начальное состояние, [nstate-1]-конечное
        short* rstates;                      //возможные состояния
автомата на данной позиции
        FST(short ns, NODE n, ...); // (массив)количество состояний
автомата, список состояний(граф переходов)
        FST(char* s, FST& fst);       // количество состояний
автомата, список состояний(граф переходов)
    };
```

Листинг 3.1. Структура конечного автомата

```
#define GRAPH_CONDITION 11,\
    FST::NODE(1,FST::RELATION('c',1)),\
    FST::NODE(1,FST::RELATION('o',2)),\
    FST::NODE(1,FST::RELATION('n',3)),\
    FST::NODE(1,FST::RELATION('d',4)),\
    FST::NODE(1,FST::RELATION('i',5)),\
    FST::NODE(1,FST::RELATION('t',6)),\
    FST::NODE(1,FST::RELATION('i',7)),\
    FST::NODE(1,FST::RELATION('o',8)),\
    FST::NODE(1,FST::RELATION('n',9)),\
    FST::NODE(1,FST::RELATION(':',10)),\
    FST::NODE()
```

Листинг 3.2 Пример реализации графа конечного автомата для токена condition

Пример графа переходов предоставляет наглядное представление о том, как каждый токен обрабатывается в контексте конечного автомата. Переходы, состояния и связанные с ними регулярные выражения являются важными компонентами лексического анализатора, обеспечивая точный и эффективный разбор выражений в исходном коде программы.

3.8. Основные структуры данных

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

```
struct Entry
    {
        char lexema;                         //лексема
        int sn;                                  //номер строки в
исходном тексте
        int idxTI;                               //индекс в ТИ
```

```
            Entry();
            Entry(char lexema, int snn, int idxti = NULLDX_TI);
      };

      struct LexTable                                    //экземпляр таблицы
лексем
      {
            int maxsize;                              //ёмкость таблицы лексем
            int size;                                 //текущий размер таблицы
лексем
            Entry* table;                             //массив строк ТЛ
      };
```

Листинг 3.3. Структура таблицы лексем

```
struct Entry
      {
            union
            {
                  int  vint;                          //значение number
                  struct
                  {
                        int len;                      //количество
символов
                        char str[STR_MAXSIZE - 1];//символы
                  } vstr;                                    //значение
строки
                  struct
                  {
                        int count;                    // количество
параметров функции
                        IDDATATYPE* types;            //типы параметров
функции
                  } params;
            } value;                                  //значение идентификатора
            int             idxfirstLE;                     //индекс в
таблице лексем
            char      id[SCOPED_ID_MAXSIZE];    //идентификатор
            IDDATATYPE      iddatatype;                     //тип данных
            IDTYPE          idtype;                         //тип
идентификатора

            Entry()                                   //конструктор без
параметров
            {
                  this->value.vint = NUM_DEFAULT;
                  this->value.vstr.len = NULL;
                  this->value.params.count = NULL;
            };
            Entry(char* id, int idxLT, IDDATATYPE datatype, IDTYPE
idtype) //конструктор с параметрами
            {
                  strncpy_s(this->id, id, SCOPED_ID_MAXSIZE - 1);
```

```
            this->idxfirstLE = idxLT;
            this->iddatatype = datatype;
            this->idtype = idtype;
        };
    };
    struct IdTable      //экземпляр таблицы идентификаторов
    {
        int maxsize;    //ёмкость таблицы идентификаторов < TI_MAXSIZE
        int size;       //текущий размер таблицы идентификаторов < maxsize
        Entry* table;   //массив строк таблицы идентификаторов
    };
```

Листинг 3.4. Структура таблицы идентификаторов

Приведенные структуры данных обеспечивают систематизацию и хранение необходимой информации для последующих этапов анализа и компиляции. Эти структуры данных играют важную роль в процессе построения комплексных протоколов работы транслятора, предоставляя полное представление о структуре исходного кода и результатов его анализа.

3.9. Структура и перечень сообщений лексического анализатора

Ошибки, возникающие в процессе трансляции программы, фиксируются в протокол, заданный входным параметрами. В случае возникновения ошибок происходит их протоколирование с номером ошибки и диагностическим сообщением. Если в процессе анализа находится более трёх ошибок, то анализ останавливается. Перечень сообщений представлен в таблице 3.3.

Таблица 3.5 – Сообщения лексического анализатора

Код	Сообщение
200	Недопустимый символ в исходном файле(-in)
201	Неизвестная последовательность символов
202	Превышен размер таблицы лексем
203	Превышен размер таблицы идентификаторов

Представленный перечень сообщений служит важным инструментом для визуальной и текстовой обратной связи, позволяя эффективно локализовать и исправить возможные проблемы в исходном коде.

3.10. Принцип обработки ошибок

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

останавливается если находится более трех ошибок.

3.10 Контрольный пример

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

4. Разработка синтаксического анализатора

4.1 Структура синтаксического анализатора

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

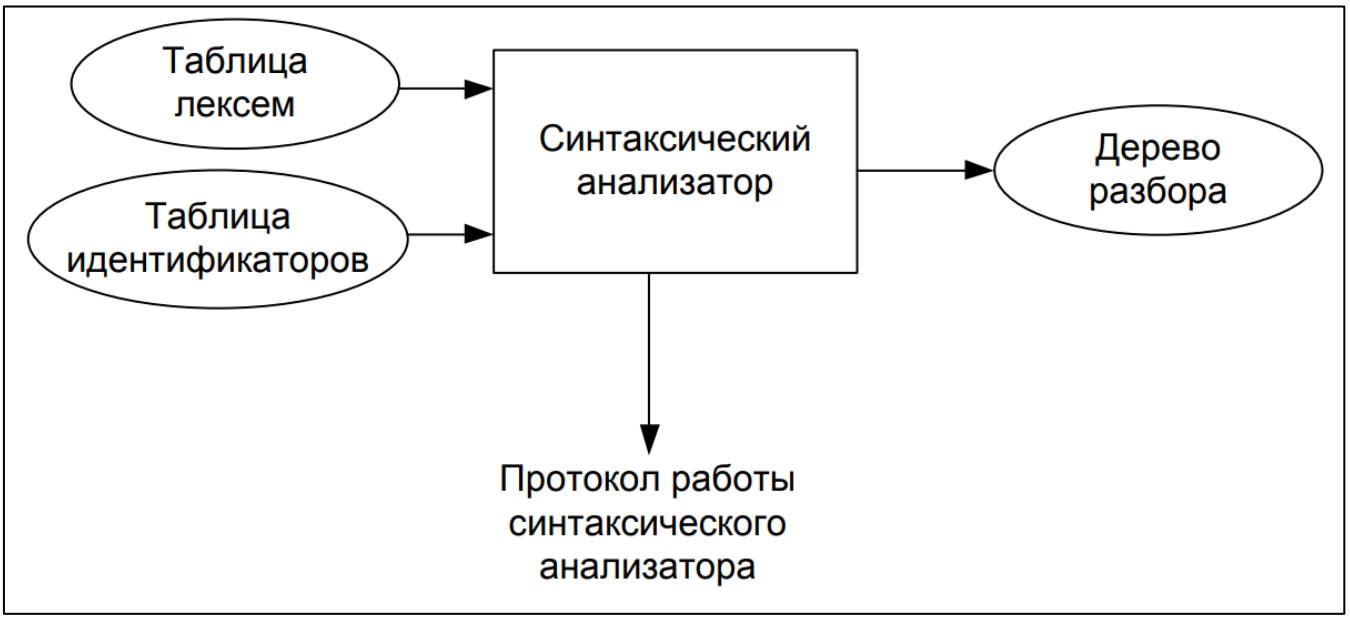

Рисунок 4.1 Структура синтаксического анализатора.

На вход синтаксическому анализатору подается таблица лексем и таблица идентификаторов. Выходной информацией является дерево разбора.

4.2 Контекстно-свободная грамматика, описывающая синтаксис языка

В синтаксическом анализаторе транслятора языка KVV-2023 используется контекстно-свободная грамматика $G = \langle T, N, P, S \rangle$, где

T – множество терминальных символов (было описано в разделе 1.2),
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 P имеют вид:

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

2) $S \to \lambda$, где $S \in N$ — начальный символ, при этом если такое правило существует, то нетерминал S не встречается в правой части правил. Описание нетерминальных символов содержится в таблице 4.1.

Таблица 4.1 Таблица правил переходов нетерминальных символов

Символ	Правила	Описание
S	S->tfiPTS S->pfiPGS S->m[K]	Стартовые правила, описывающее общую структуру программы
P	P->(E) P->()	Правила для параметров объявляемых функций
T	T->[eV;] T->[KeV;]	Правила для тела функций
G	G->[e;] G->[Ke;]	Правила для тела процедур
E	E->ti,E E->ti	Правила для списка параметров функции
F	F->(N) F->()	Правила для вывозов функций(в т.ч. и в выражениях)
N	N->i N>l N->l,N N->I,N	Правила для параметров вызываемых функций
R	R->rY# R>wY# R>cY# R->rYwY# R->wYrY#	Правила составления цикла/условного оператора
Z	Z->iLi Z->iLl Z->lli	Правила для условия цикла/условного оператора
L	L->< L->> L->!	Правила для логических операторов
A	A->+ A->- A->* A->/ A->{ A->}	Правила для арифметических и свдиговых операторов

Окончание таблицы 4.1

Символ	Правила	Описание
V	V->l V->i V->q	Правила для простых выражений
Y	Y->[X]	Правила для тела цикла/условного выражения
W	W->l W->i W->(W) W->(W)AW W->iF W->iAW W->lAW W->iFAW	Правила для сложных выражений
K	K->nti=V;K K->nti;K K->i=W;K K->oV;K K->^;K K->&Z#RK K->iF;K K->nti=V; K->nti; K->i=W; K->oV; K->^; K->&Z#R K->iF;	Программные конструкции
X	X->i=W;X X->oV;X X->^;X X->iF;X X->i=W; X->oV; X->^; X->iF;	Программные конструкции внутри цикла/условного оператора

Такой формат грамматики обеспечивает структурированное и однозначное понимание синтаксических конструкций языка KVV-2023. На основе этой грамматики синтаксический анализатор строит синтаксическое дерево, представляющее собой абстрактное представление структуры программы.

4.3 Построение конечного магазинного автомата

Конечный автомат с магазинной памятью представляет собой семерку

$M = \langle Q, V, Z, \delta, q_0, z_0, F \rangle$. Подробное описание компонентов магазинного автомата 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 3.1 и 4.1.
Z	Алфавит специальных магазинных символов	Алфавит магазинных символов содержит стартовый символ и маркер дна стека (представляет из себя 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. Представляется в виде стартового правила грамматики
z_0	Начальное состояние магазина автомата	Символ маркера дна стека $
F	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

Пример разбора цепочки языка находится в приложении В.

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 KVV-2023. Данные структуры приведены в приложении В.

4.5 Описание алгоритма синтаксического разбора

Принцип работы автомата следующий:

- В магазин записывается стартовый символ;
- На основе полученных ранее таблиц формируется входная лента;
- Запускается автомат;
- Выбирается цепочка, соответствующая нетерминальному символу, записывается в магазин в обратном порядке;
- 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;
- Если в магазине встретился нетерминал, переходим пункту 4;
- Если наш символ достиг дна стека, и лента в этот момент пуста, то синтаксический анализ выполнен успешно. Иначе генерируется исключение.

4.6. Параметры синтаксического анализатора и режимы его работы

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

4.7. Структура и перечень сообщений синтаксического анализатора

Перечень сообщений синтаксического анализатора представлен в таблице 4.3.

Таблица 4.3 – Сообщение синтаксического анализатора

Код	Сообщение
600	Неверная структура программы
601	Не найден список параметров функции
602	Ошибка в теле функции
603	Ошибка в теле процедуры
604	Ошибка в списке параметров функции
605	Ошибка в вызове функции/выражении
606	Ошибка в списке фактических параметров функции
607	Ошибка при конструировании цикла/условного выражения
608	Ошибка в теле цикла/условного выражения
609	Ошибка в условии цикла/условного выражения
610	Неверный условный оператор
611	Неверный арифметический оператор
612	Неверное выражение. Ожидаются только идентификаторы/литералы
613	Ошибка в арифметическом выражении
614	Недопустимая синтаксическая конструкция
615	Недопустимая синтаксическая конструкция в теле цикла/условного выражения

Представленный перечень сообщений служит важным инструментом для визуальной и текстовой обратной связи, позволяя эффективно локализовать и исправить возможные проблемы.

4.8. Принцип обработки ошибок

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

4.9. Контрольный пример

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

5 Разработка семантического анализатора

5.1 Структура семантического анализатора

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

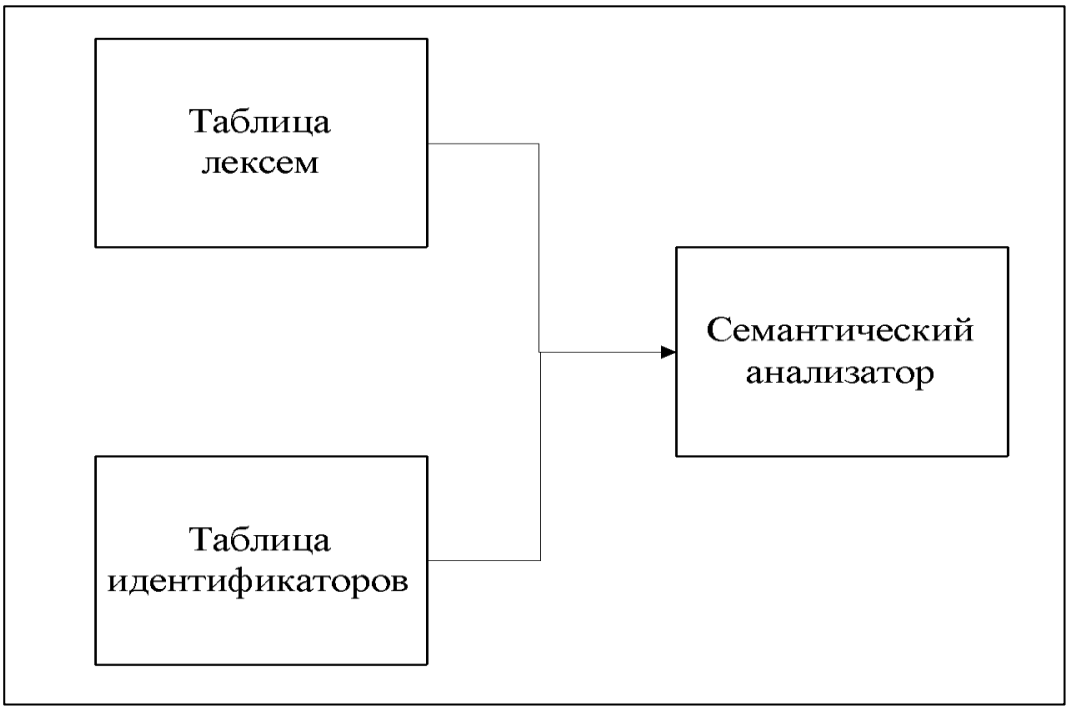

Рисунок 5.1. Структура семантического анализатора

Семантический анализатор гарантирует корректность и согласованность семантической структуры программы на языке KVV-2023.

5.2 Функции семантического анализатора

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

5.3 Структура и перечень сообщений семантического анализатора

Сообщения, формируемые семантическим анализатором, представлены в таблице 5.1.

Таблица 5.1 – Перечень сообщений семантического анализатора

Код	Сообщение
300	Необъявленный идентификатор
301	Отсутствует точка входа main
302	Обнаружено несколько точек входа main
303	В объявлении не указан тип идентификатора
304	В объявлении отсутствует ключевое слово new
305	Попытка переопределения идентификатора
306	Превышено максимальное количество параметров функции
307	Слишком много параметров в вызове
309	Несовпадение типов передаваемых параметров
310	Использование пустого строкового литерала недопустимо
311	Обнаружен символ “. Возможно, не закрыт строковый литерал
312	Превышен размер строкового литерала
313	Недопустимы целочисленный литерал
314	Типы данных в выражении не совпадают
315	Тип функции и возвращаемого значения не совпадают
316	Недопустимое строковое выражение справа от знака ‘=’
317	Неверное условное выражение

Представленный перечень сообщений служит важным инструментом для визуальной и текстовой обратной связи, позволяя эффективно локализовать и исправить возможные проблемы.

5.4 Принцип обработки ошибок

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

5.5 Контрольный пример

Демонстрация ошибок, диагностируемых семантическим анализатором приведена в разделе 8.4.

6. Вычисление выражений

6.1 Выражения, допускаемые языком

В языке KVV-2023 допускаются вычисления выражений целочисленного типа данных с поддержкой вызова функций внутри выражений. Приоритет операций представлен на таблице 6.1.

Таблица 6.1. Приоритеты операций

Операция	Значение приоритета
( )	3
*	2
/	2
+	1
-	1
}	0
{	0

Такая система приоритетов операций является важным элементом языка, обеспечивая корректное вычисление выражений и предоставляя контроль над порядком операций. Это способствует более наглядному и предсказуемому написанию программ, облегчая понимание и поддержание кода.

6.2 Польская запись и принцип её построения

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

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

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

6.3 Программная реализация обработки выражений

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

6.4 Контрольный пример

Пример преобразования выражений из контрольных примеров к обратной польской записи представлен в таблице 6.2. В приведенном примере в квадратных скобках указывается номер идентификатора/литерала в таблице идентификаторов. Преобразование выражений в формат польской записи необходимо для построения более простых алгоритмов, их вычисления и преобразования к ассемблерному коду. В приложении Г приведены изменённые таблицы лексем и идентификаторов, отображающие результаты преобразования выражений в формат польской записи.

Таблица 6.2. Преобразование выражений к ПОЛИЗ

Выражение	Обратная польская запись для выражения
i[2]=(((l[3]+l[4])-i[0])*l[5])/l[6];	i[2]=l[3]l[4]+i[0]-l[5]*l[6]/
i[23]=(i[23]+l[26])*l[26]	i[23]=i[23]l[26]+l[26]*
i[3]=(((l[4]+l[5])-i[0])*l[6])	i[3]=l[4]l[5]+i[0]-l[6]*

Преобразование выражений в обратную польскую запись позволяет строить более простые алгоритмы вычисления и облегчает последующее преобразование к ассемблерному коду. Контрольный пример в таблице 6.2 демонстрирует успешное преобразование выражений в польскую запись для различных случаев. Это подтверждает функциональность реализованного алгоритма и его способность обрабатывать разнообразные выражения.

7. Генерация кода

7.1 Структура генератора кода

В языке KVV-2023 генерация кода является заключительным этапом трансляции. Генератор принимает на вход таблицы лексем и идентификаторов, полученные в результате лексического анализа. В соответствии с таблицей лексем строится выходной файл на языке ассемблера, который будет являться результатом работы транслятора. В случае возникновения ошибок генерация кода не будет осуществляться. Структура генератора кода KVV-2023 представлена на рисунке 7.1.

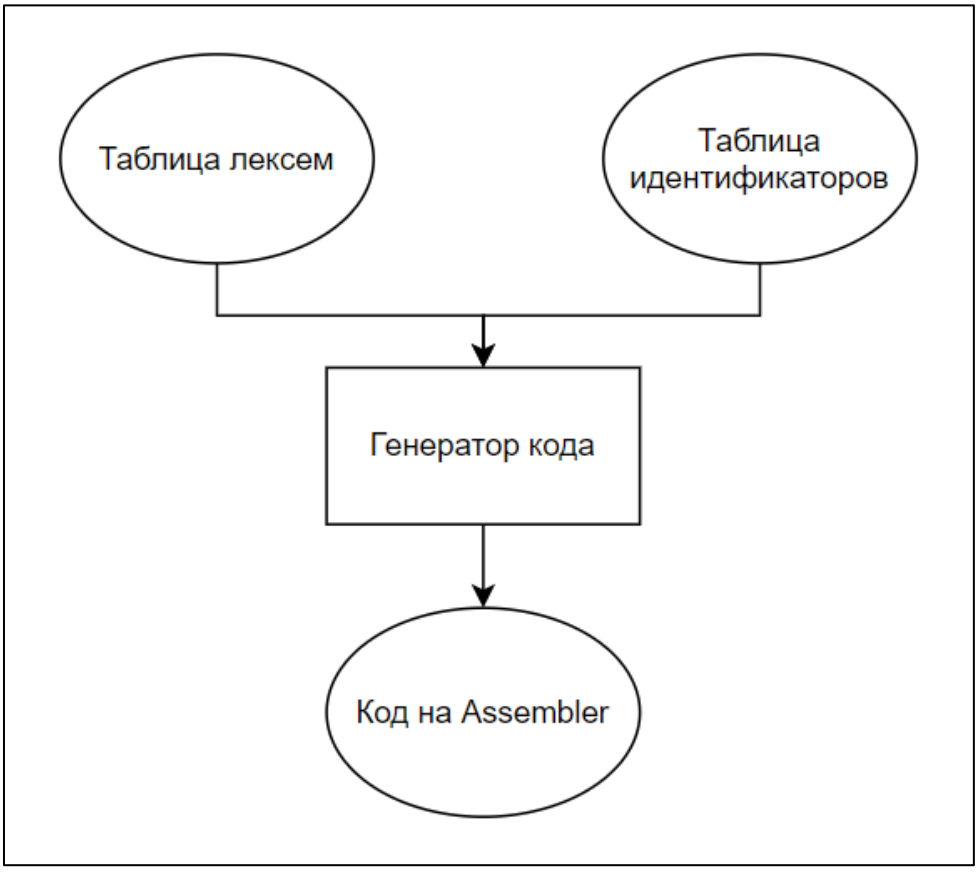

Рисунок 7.1 – Структура генератора кода

Структура генератора кода KVV-2023, представленная на рисунке 7.1, указывает на последовательность операций, выполняемых на этом этапе.

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 KVV-2023 и на языке ассемблера приведены в таблице 7.1.

Таблица 7.1 – Соответствия типов идентификаторов языка KVV-2023 и языка ассемблера

Тип идентификатора на языке KVV-2023	Тип идентификатора на языке ассемблера	Пояснение
number	sdword	Хранит целочисленный тип данных.
string	dword	Хранит указатель на начало строки.
l	byte dword	Хранит значение строкового литерала; Хранит значение целочисленного литерала.

Важным моментом является соответствие между типами данных идентификаторов на уровне KVV-2023 и соответствующими типами на языке ассемблера, что представлено в таблице 7.1. Такое соответствие обеспечивает правильное распределение и хранение данных в сегментах памяти ассемблера, что является важным шагом при генерации кода на этапе трансляции.

7.3 Статическая библиотека

В языке KVV-2023 предусмотрена статическая библиотека. Статическая библиотека содержит функции, написанные на языке C++ и подключается автоматически на начальном этапе генерации кода используя директиву “**includelib**”. Объявление функций статической библиотеки генерируется автоматически в коде ассемблера. Функции статической библиотеки описаны в разделе 1.18.

7.4 Особенности алгоритма генерации кода

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

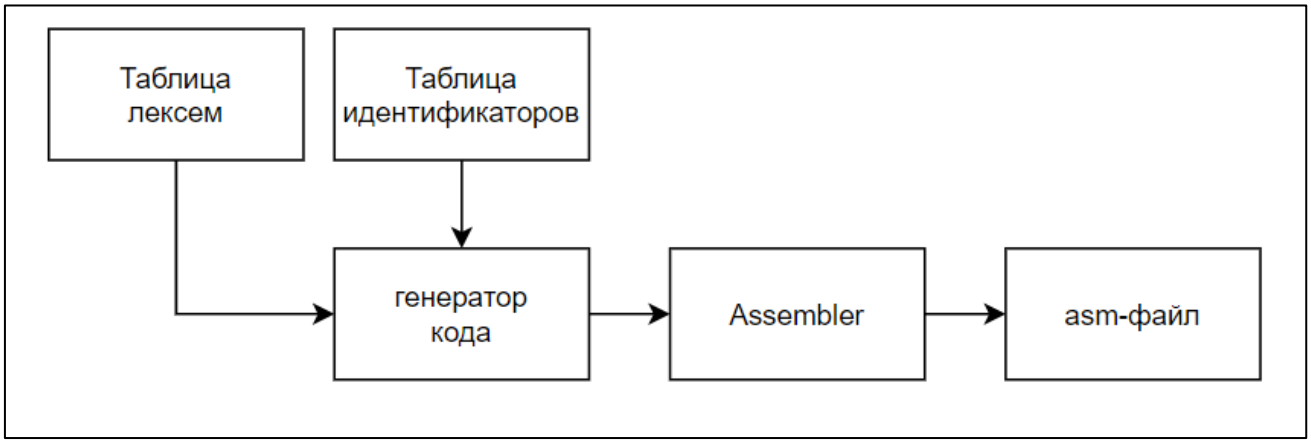

Рисунок 7.2 – Структура генератора кода

Алгоритм представляет собой систему генерации в язык Assembler для языка программирования KVV-2023. На вход генератору подаются таблицы лексем и идентификаторов. Генератор проходит по таблице лексем, определяя тип лексемы и вызывая соответствующие функции генерации кода. Перечень реализованных для генерации кода функций представлена в таблице 7.2.

Таблица 7.2 – Перечень функций для генерации кода

Функция	Описание
CodeGeneration	Осуществляет генерацию кода на ассемблере для всего программного кода. В качестве параметров принимает таблицы лексем и идентификаторов.
StartFillVector	Генерирует начальный набор строк для файла с кодом на ассемблере, включая секции ‘.const’, ‘.data’, ‘.code’. В качестве параметров принимает таблицу лексем
itoS	Преобразует целое число в строку. В качестве параметра принимает целое число.
genFunctionCode	Генерирует код для объявления функции. В качестве параметров принимает таблицу лексем, индекс лексемы, представляющей объявление функции, имя объявляемой функции и ее параметров.
genCallFuncCode	Генерирует код вызова функции. В качестве параметров принимает таблицу лексем и индекс лексемы представляющей вызов функции.
genEqualCode	Генерирует код для операции присваивания и вычисления выражений. В качестве параметров принимает таблицу лексем и индекс лексемы, представляющей операцию присваивания.
genConditionCode	Генерирует код для условного оператора. В качестве параметров принимает таблицу лексем, индекс лексемы, представляющей условный оператор, строку для хранения кода цикла, если условие в цикле.
genExitCode	Генерирует код завершения функции. В качестве параметров принимает таблицу лексем, индекс лексемы представляющей завершение функции, имя текущей функции и количество параметров функции.

Разделение алгоритма на отдельные функции для генерации различных конструкций способствует модульности и пониманию каждой части генератора кода.

7.5 Параметры, управляющие генерацией кода

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

8. Тестирование транслятора

8.1 Тестирование проверки на допустимость символов

В языке KVV-2023 не разрешается использовать запрещённые входным алфавитом символы. Результат использования запрещённого символа показан в таблице 8.1.

Таблица 8.1 - Тестирование проверки на допустимость символов

Исходный код	Диагностическое сообщение
main [ё]	Ошибка N200: Лексическая ошибка: Недопустимый символ в исходном файле(-in) Строка: 2 Позиция в строке: 2

Это правило служит для поддержания стандартов безопасности и целостности исходного кода, предотвращая возможные ошибки и недоразумения при компиляции.

8.2 Тестирование лексического анализатора

На этапе лексического анализа в языке KVV-2023 могут возникнуть ошибки, описанные в пункте 3.7. Результаты тестирования лексического анализатора показаны в таблице 8.2.

Таблица 8.2 - Тестирование лексического анализатора

Исходный код	Диагностическое сообщение
main [new number x11;]	Ошибка N201: Лексическая ошибка: Неизвестная последовательность символов Строка: 3

Эти ошибки служат важным инструментом для выявления и коррекции проблем в исходном коде ещё на ранних этапах компиляции.

8.3 Тестирование синтаксического анализатора

На этапе синтаксического анализа в языке KVV-2023 могут возникнуть ошибки, описанные в пункте 4.7. Результаты тестирования синтаксического анализатора показаны в таблице 8.3.

Таблица 8.3 - Тестирование синтаксического анализатора

Исходный код	Диагностическое сообщение
main new number x;]	Ошибка 600: строка 1, Синтаксическая ошибка: Неверная структура программы
string function fi([] main[]	Ошибка 601: строка 1, Синтаксическая ошибка: Не найден список параметров функции
string function fi() [newline; write]	Ошибка 602: строка 1, Синтаксическая ошибка: Ошибка в теле функции

Окончание таблицы 8.3

Исходный код	Диагностическое сообщение
main[]	
procedure function fi() [newline write;] main[]	Ошибка 603: строка 1, Синтаксическая ошибка: Ошибка в теле процедуры
procedure function fi(number number)[] main[]	Ошибка 604: строка 1, Синтаксическая ошибка: Ошибка в списке параметров функции
string function fi(number x)[return 3;] main [newline;fi(5,5; newline;]	Ошибка 605: строка 2, Синтаксическая ошибка: Ошибка в вызове функции/выражении
string function fi(number x)[return 3;] main [newline;fi(5,5,5 5);]	Ошибка 606: строка 2, Синтаксическая ошибка: Ошибка в списке фактических параметров функции
main [new number x; condition: x > 2 # cycle #]	Ошибка 607: строка 1, Синтаксическая ошибка: Ошибка при конструировании цикла/условного выражения
main [new number x; condition: x > 2 # cycle #]	Ошибка 608: строка 1, Синтаксическая ошибка: Ошибка в теле цикла/условного выражения
main [condition: 1 = 2 #]	Ошибка 609: строка 1, Синтаксическая ошибка: Ошибка в условии цикла/условного выражения
main [condition: 1 = 2 #]	Ошибка 610: строка 1, Синтаксическая ошибка: Неверный условный оператор
main [new number x; x = x ! x;]	Ошибка 611: строка 1, Синтаксическая ошибка: Неверный арифметический оператор
main [new number x; write new;]	Ошибка 612: строка 1, Синтаксическая ошибка: Неверное выражение. Ожидаются только идентификаторы/литералы
main [new number x; x = 1 ++;]	Ошибка 613: строка 1, Синтаксическая ошибка: Ошибка в арифметическом выражении
main [newline; 4;]	Ошибка 614: строка 1, Синтаксическая ошибка: Недопустимая синтаксическая конструкция
main [new number a; condition: a < 3 # istrue [newline; 3;] #]	Ошибка 615: строка 1, Синтаксическая ошибка: Недопустимая синтаксическая конструкция в теле цикла/условного выражения

Эти ошибки помогают выявить структурные и синтаксические проблемы в программе ещё на этапе компиляции, что обеспечивает более быструю и точную диагностику.

8.4 Тестирование семантического анализатора

Семантический анализ в языке KVV-2023 содержит множество проверок по семантическим правилам, описанным в пункте 1.16. Итоги тестирования семантического анализатора на корректное обнаружение семантических ошибок приведены в таблице 8.4.

Таблица 8.4 - Тестирование семантического анализатора

Исходный код	Диагностическое сообщение
main [a = 1]	Ошибка N300: Семантическая ошибка: Необъявленный идентификатор Строка: 1
string function fi()[]	Ошибка N301: Семантическая ошибка: Отсутствует точка входа main Строка: 0
main[] main[]	Ошибка N302: Семантическая ошибка: Обнаружено несколько точек входа main Строка: 0
main[a = 1;]	Ошибка N304: Семантическая ошибка: В объявлении отсутствует ключевое слово new Строка: 1
main[new number t; new string t;]	Ошибка N305: Семантическая ошибка: Попытка переопределения идентификатора Строка: 3
procedure function fi()[] main[fi(“a”,”b”,”c”,”d”]	Ошибка N307: Семантическая ошибка: Слишком много параметров в вызове Строка: 1
string function fi(string x, string y, string z, string s) main[]	Ошибка N306: Семантическая ошибка: Превышено максимальное количество параметров функции Строка: 1
string function fi(string x)[return "a";] main[fi("a", "b");]	Ошибка N308: Семантическая ошибка: Кол-во ожидаемых функцией и передаваемых параметров не совпадают Строка: 2
string function fi(string x)[return "a";] main[fi("a", "b");]	Ошибка N309: Семантическая ошибка: Несовпадение типов передаваемых параметров Строка: 2
main[new string x="";]	Ошибка N310: Семантическая ошибка: Использование пустого строкового литерала недопустимо Строка: 1
main[new string x=";]	Ошибка N311: Семантическая ошибка: Обнаружен символ "". Возможно, не закрыт строковый литерал Строка: 1
main[new number x=99999999999999999;]	Ошибка N313: Семантическая ошибка: Недопустимый целочисленный литерал Строка: 1
main[new number x; x = 5 + "abc";]	Ошибка N314: Семантическая ошибка: Типы данных в выражении не совпадают Строка: 1
string function fi()[return 5;] main[newline;]	Ошибка N315: Семантическая ошибка: Тип функции и возвращаемого значения не совпадают Строка: 1
main[new string x; x = "abc" + "d";]	Ошибка N316: Семантическая ошибка: Недопустимое строковое выражение справа от знака '=' Строка: 1

Окончание таблицы 8.4

Исходный код	Диагностическое сообщение
main [condition: "string"& 6# istrue[write "string";]]	Ошибка N317: Семантическая ошибка: Неверное условное выражение Строка: 1
main[new number a =5; a = a/0; write a;]	Ошибка N318: Семантическая ошибка: Деление на ноль Строка: 4

Эти ошибки помогают выявить и предотвратить нарушения семантических правил в программе, что способствует более безопасному и надежному исполнению программы на языке KVV-2023.

Заключение

В ходе выполнения курсовой работы был разработан транслятор и генератор кода для языка программирования KVV-2023. Таким образом, были выполнены задачи данной курсовой работы:

- Сформулирована спецификация языка KVV-2023;
- Разработан дизайн языка KVV-2023;
- Разработаны конечные автоматы и важные алгоритмы на их основе для эффективной работы лексического анализатора;
- Осуществлена программная реализация лексического анализатора, распознающего допустимые цепочки спроектированного языка;
- Разработана контекстно-свободная, приведённая к нормальной форме Грейбах, грамматика для описания синтаксически верных конструкций языка;
- Осуществлена программная реализация синтаксического анализатора;
- Разработан семантический анализатор, осуществляющий проверку используемых инструкций на соответствие логическим правилам;
- Разработан транслятор кода на язык ассемблера;
- Проведено тестирование всех вышеперечисленных компонентов.
- Окончательная версия языка KVV-2023 включает:
- 2 типа данных;
- Поддержка операторов вывода и перевода строки;
- Возможность вызова функций стандартной библиотеки;
- Наличие 4 арифметических операторов для вычисления выражений;
- Наличие сдвиговых операций;
- Поддержка функций, процедур, операторов цикла и условия;
- Структурированная и классифицированная система для обработки ошибок пользователя.

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

Список использованных источников

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

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

3. Прата, С. Язык программирования C++. Лекции и упражнения / С. Прата. – М., 2006.

4. Страуструп, Б. Принципы и практика использования C++ / Б. Страуструп – 2009.

5. Волкова И.А., Руденко Т.В. Формальные языки и грамматики. Элементы теории трансляции. - М.: Диалог-МГУ, 1999.

6. Польская запись [Электронный ресурс]. – Режим доступа: https://ru.wikipedia.org/wiki/Польская_запись. – Дата доступа 02.12.2023.

7. Форма Бэкуса-Наура [Электронный ресурс]. – Режим доступа: https://ru.wikipedia.org/wiki/Форма_Бэкуса_-_Наура. – Дата доступа 04.12.2023.

Приложение А

Листинг 1 – Исходный код программы на языке KVV-2023

```
number function min(number x, number y)
[
        new number res;
        condition: x < y #
        istrue [res = x;]
        isfalse [res = y;]#
        return res;
]
procedure function stand (string a, string b)
[
        new number k;
        k = lenght(a)+1;
       write "Len + 1:";
        write k;
        newline;
        new string str;
        str = concat(a,b);
       write "concat:";
        write str;
        newline;

        return;
]

main
[
        new number x = 9;
        new number y = -9;
        new string strx = "Just";
        new string stry = "string";
       new string strz = "70";
       new number e;
       write "from string to number:";
       e = atoii(strz);
       write e;
       newline;
       new number result;
       result = x{1{2;
       write "sdvig left:";
       write result;
       newline;
        new number t;
```

```
        t = min (x,y);
      write t;
        newline;
        new number ab = 3;
      new number d = 0q120;
        condition: ab ! 52 #
        cycle [
                write ab;
                write " ";
                ab = (ab + 2)*2;
        ]#
        write " after cycle ";
        write ab;
        newline;
        stand(strx, stry);
]
```

Приложение Б

Листинг 1 Таблица идентификаторов контрольного примера

```
|  N  | СТРОКА В ТЛ | ТИП ИДЕНТИФИКАТОРА | ИМЯ| ЗНАЧЕНИЕ (ПАРАМ)
|   0 |      2    |  number    function |              min | P0:NUMBER | P1:NUMBER |
|   1 |      5    |  number  parameter |              minx |
|   2 |      8    |  number  parameter |              miny |
|   3 |     13    |  number    variable |            minres |0
|   4 |     41    |    proc     function |            stand | P0:STRING | P1:STRING |
|   5 |     44    | string    parameter |            standa |
|   6 |     47    | string    parameter |            standb |
|   7 |     52    |  number    variable |            standk |0
|   8 |     56    |  number    LIB FUNC |           lenght | P0:STRING |
|   9 |     68    | string     variable |          standstr |[0]
|  10 |     72    | string     LIB FUNC |           concat | P0:STRING | P1:STRING|
|  11 |     91    |  number    variable |            mainx |0
|  12 |     93    |  number     literal |           LTRL1 |9
|  13 |     97    |  number    variable |            mainy |0
|  14 |     99    |  number     literal |           LTRL2 |-9
|  15 |    103    | string     variable |          mainstrx |[0]
|  16 |    105    | string      literal |           LTRL3 |[4]Just
|  17 |    109    | string     variable |          mainstry |[0]
|  18 |    111    | string      literal |           LTRL4 |[6]string
|  19 |    115    |  number    variable |            maint |0
|  20 |    133    |  number    variable |            mainab |0
|  21 |    135    |  number     literal |           LTRL5 |3
|  22 |    140    |  number     literal |           LTRL6 |52
|  23 |    148    | string      literal |           LTRL7 |[1]
|  24 |    155    |  number     literal |           LTRL8 |2
```

Листинг 2 Таблица лексем после контрольного примера

N	ЛЕКСЕМА	СТРОКА	ИНДЕКС В ТИ
0	t	1	
1	f	1	
2	i	1	0
3	(	1	
4	t	1	
5	i	1	1
6	,	1	
7	t	1	
8	i	1	2
9	)	1	
10	[	2	
11	n	3	
12	t	3	
13	i	3	3
14	;	3	
15	?	4	
16	i	4	1
17	<	4	
18	i	4	2
19	#	4	
20	w	5	
21	[	5	
22	i	5	3
23	=	5	
24	i	5	1
25	;	5	
26	]	5	
27	r	6	
28	[	6	
29	i	6	3
30	=	6	
31	i	6	2
32	;	6	
33	]	6	
34	#	6	
35	e	7	
36	i	7	3
37	;	7	
38	]	8	
39	p	9	
40	f	9	
41	i	9	4
42	(	9	
43	t	9	
44	i	9	5
45	,	9	
46	t	9	
47	i	9	6
48	)	9	
49	[	10	
50	n	11	
51	t	11	
52	i	11	7
53	;	11	
54	i	12	7
55	=	12	
56	i	12	8
57	(	12	
58	i	12	5
59	)	12	
60	;	12	
61	o	13	
62	i	13	7
63	;	13	
64	^	14	
65	;	14	
66	n	15	
67	t	15	
68	i	15	9
69	;	15	
70	i	16	9
71	=	16	
72	i	16	10
73	(	16	
74	i	16	5
75	,	16	
76	i	16	6
77	)	16	
78	;	16	
79	o	17	
80	i	17	9
81	;	17	
82	^	18	
83	;	18	
84	e	19	
85	;	19	
86	]	20	
87	m	21	
88	[	22	
89	n	23	
90	t	23	
91	i	23	11
92	=	23	
93	l	23	12
94	;	23	
95	n	24	
96	t	24	
97	i	24	13
98	=	24	

99	l	24	14
100	;	24	
101	n	25	
102	t	25	
103	i	25	15
104	=	25	
105	l	25	16
106	;	25	
107	n	26	
108	t	26	
109	i	26	17
110	=	26	
111	l	26	18
112	;	26	
113	n	27	
114	t	27	
115	i	27	19
116	;	27	
117	i	28	19
118	=	28	
119	i	28	0
120	(	28	
121	i	28	11
122	,	28	
123	i	28	13
124	)	28	
125	;	28	
126	o	29	
127	i	29	19
128	;	29	
129	^	30	
130	;	30	
131	n	31	
132	t	31	
133	i	31	20
134	=	31	
135	l	31	21
136	;	31	
137	?	32	
138	i	32	20
139	!	32	
140	l	32	22
141	#	32	
142	c	33	
143	[	33	
144	o	34	
145	i	34	20
146	;	34	
147	o	35	
148	l	35	23
149	;	35	
150	i	36	20
151	=	36	
152	(	36	
153	i	36	20
154	+	36	
155	l	36	24
156	)	36	
157	*	36	
158	l	36	24
159	;	36	
160	]	37	
161	#	37	
162	o	38	
163	l	38	23
164	;	38	
165	o	39	
166	i	39	20
167	;	39	
168	^	40	
169	;	40	
170	i	41	4
171	(	41	
172	i	41	15
173	,	41	
174	i	41	17
175	)	41	
176	;	41	
177	]	42	

Приложение В

Листинг 1 Грамматика языка KVV-2023

```
reibach greibach(NS('S'), TS('$'), 16,
             Rule(NS('S'), GRB_ERROR_SERIES, 3,       // Неверная структура программы
                    Rule::Chain(6, TS('t'), TS('f'), TS('i'), NS('P'), NS('T'), NS('S')),
                    Rule::Chain(6, TS('p'), TS('f'), TS('i'), NS('P'), NS('G'), NS('S')),
                    Rule::Chain(4, TS('m'), TS('['), NS('K'), TS(']'))
             ),

             Rule(NS('T'), GRB_ERROR_SERIES + 2, 2,   // Ошибка в теле функции
                    Rule::Chain(5, TS('['), TS('e'), NS('V'), TS(';'), TS(']')),
                    Rule::Chain(6, TS('['), NS('K'), TS('e'), NS('V'), TS(';'), TS(']'))
             ),

             Rule(NS('G'), GRB_ERROR_SERIES + 3, 2,   // Ошибка в теле процедуры
                    Rule::Chain(4, TS('['), TS('e'), TS(';'), TS(']')),
                    Rule::Chain(5, TS('['), NS('K'), TS('e'), TS(';'), TS(']'))

             Rule(NS('P'), GRB_ERROR_SERIES + 1, 2,   // Не найден список параметров функции
                    Rule::Chain(3, TS('('), NS('E'), TS(')')),
                    Rule::Chain(2, TS('('), TS(')'))
             ),

             Rule(NS('E'), GRB_ERROR_SERIES + 4, 2,   // Ошибка в списке параметров функции
                    Rule::Chain(4, TS('t'), TS('i'), TS(','), NS('E')),
                    Rule::Chain(2, TS('t'), TS('i'))
             ),

             Rule(NS('F'), GRB_ERROR_SERIES + 5, 2,   // Ошибка в вызове функции

                    Rule::Chain(3, TS('('), NS('N'), TS(')')),
                    Rule::Chain(2, TS('('), TS(')'))
             ),

             Rule(NS('N'), GRB_ERROR_SERIES + 6, 4,   // Ошибка в списке параметров функции
                    Rule::Chain(1, TS('i')),
                    Rule::Chain(1, TS('l')),
                    Rule::Chain(3, TS('i'), TS(','), NS('N')),
                    Rule::Chain(3, TS('l'), TS(','), NS('N'))
             ),

             Rule(NS('R'), GRB_ERROR_SERIES + 7, 5,   // Ошибка при констуировании
цикла/условного выражения
                    Rule::Chain(3, TS('r'), NS('Y'), TS('#')),
                    Rule::Chain(3, TS('w'), NS('Y'), TS('#')),
                    Rule::Chain(3, TS('c'), NS('Y'), TS('#')),
                    Rule::Chain(5, TS('r'), NS('Y'), TS('w'), NS('Y'), TS('#')),
                    Rule::Chain(5, TS('w'), NS('Y'), TS('r'), NS('Y'), TS('#'))
             ),

             Rule(NS('Y'), GRB_ERROR_SERIES + 8, 1,   // Ошибка в теле цикла/условного
выражения
                    Rule::Chain(3, TS('['), NS('X'), TS(']'))
             ),

             Rule(NS('Z'), GRB_ERROR_SERIES + 9, 3,   // Ошибка в условии цикла/условного
выражения
                    Rule::Chain(3, TS('i'), NS('L'), TS('i')),
                    Rule::Chain(3, TS('i'), NS('L'), TS('l')),
                    Rule::Chain(3, TS('l'), NS('L'), TS('i'))
             ),
```

Листинг 1(продолжение) Грамматика языка KVV-2023

```
Rule(NS('L'), GRB_ERROR_SERIES + 10, 4,// Неверный условный оператор
                        Rule::Chain(1, TS('<')),
                        Rule::Chain(1, TS('>')),
                        Rule::Chain(1, TS('&')),
                        Rule::Chain(1, TS('!'))
                ),

                Rule(NS('A'), GRB_ERROR_SERIES + 11, 4,  // Неверный арифметический оператор
                        Rule::Chain(1, TS('+')),
                        Rule::Chain(1, TS('-')),
                        Rule::Chain(1, TS('*')),
                        Rule::Chain(1, TS('/'))
                ),

                Rule(NS('V'), GRB_ERROR_SERIES + 12, 2,  // Неверное выражение.
        Rule::Chain(1, TS('l')),
                        Rule::Chain(1, TS('i'))
                ),

                Rule(NS('W'), GRB_ERROR_SERIES + 13, 8,// Ошибка в арифметичском выражении
                        Rule::Chain(1, TS('i')),
                        Rule::Chain(1, TS('l')),
                        Rule::Chain(3, TS('('), NS('W'), TS(')')),
                        Rule::Chain(5, TS('('), NS('W'), TS(')'), NS('A'), NS('W')),
                        Rule::Chain(2, TS('i'), NS('F')),
                        Rule::Chain(3, TS('i'), NS('A'), NS('W')),
                        Rule::Chain(3, TS('l'), NS('A'), NS('W')),
                        Rule::Chain(4, TS('i'), NS('F'), NS('A'), NS('W'))
                ),

                Rule(NS('K'), GRB_ERROR_SERIES + 14, 14,// Недопустимая синтаксическая конструкция
                        Rule::Chain(7, TS('n'), TS('t'), TS('i'), TS('='), NS('V'), TS(';'),
NS('K')),     // декларация + присваивание
                Rule::Chain(5, TS('n'), TS('t'), TS('i'), TS(';'), NS('K')), // декларация
                Rule::Chain(5, TS('i'), TS('='), NS('W'), TS(';'), NS('K')), // присваивание
                Rule::Chain(4, TS('o'), NS('V'), TS(';'), NS('K')),          // вывод
                Rule::Chain(3, TS('^'), TS(';'), NS('K')),                   // перевод строки
                        Rule::Chain(5, TS('?'), NS('Z'), TS('#'), NS('R'), NS('K')), // condition
                        Rule::Chain(4, TS('i'), NS('F'), TS(';'), NS('K')),    // вызов функции
                        Rule::Chain(6, TS('n'), TS('t'), TS('i'), TS('='), NS('V'), TS(';')),
        // декларация + присваивание
                        Rule::Chain(4, TS('i'), TS('='), NS('W'), TS(';')),    // присваивание
                        Rule::Chain(4, TS('n'), TS('t'), TS('i'), TS(';')),    // декларация
                        Rule::Chain(3, TS('o'), NS('V'), TS(';')),             // вывод
                        Rule::Chain(2, TS('^'), TS(';')),                      // перевод строки
                        Rule::Chain(4, TS('?'), NS('Z'), TS('#'), NS('R')),    // condition
                        Rule::Chain(3, TS('i'), NS('F'), TS(';'))              // вызов функции
                ),
                Rule(NS('X'), GRB_ERROR_SERIES + 15, 8,                        // Недопустимая
синтаксическая конструкция в теле цикла/условного выражения
                        Rule::Chain(5, TS('i'), TS('='), NS('W'), TS(';'), NS('X')), //присваивание
                        Rule::Chain(4, TS('o'), NS('V'), TS(';'), NS('X')),    // вывод
                        Rule::Chain(3, TS('^'), TS(';'), NS('X')),             // перевод строки
                        Rule::Chain(4, TS('i'), NS('F'), TS(';'), NS('X')),    // вызов функции
                        Rule::Chain(4, TS('i'), TS('='), NS('W'), TS(';')),    // присваивание
                        Rule::Chain(3, TS('o'), NS('V'), TS(';')),      // вывод
                        Rule::Chain(2, TS('^'), TS(';')),               // перевод строки
                        Rule::Chain(3, TS('i'), NS('F'), TS(';'))       // вызов функции
                )
        );
```

Листинг 2 Структура магазинного автомата

```
struct Mfst {                       //магазинный автомат
              enum RC_STEP          //шаг автомата
              {
                     NS_OK,               //найдено правило и цепочка, цепочка записана в стек
                     NS_NORULE,           //не найдено правило грамматики (ошибки в грамматике)
                     NS_NORULECHAIN,//не найдена подходящая цепочка правила
                     NS_ERROR,     //неизвестный нетерминальный символ грамматики
                     TS_OK, //текущий символ ленты == вершине стека, продвинулась лента
                     TS_NOK,//текущий символ ленты != вершине стека, восстановлено состояние
                     LENTA_END,    //текущая позиция ленты >= lenta_size
                     SURPRISE      //неожиданный код возврата ( ошибка в step)
              };
              struct MfstDiagnosis       //диагностика
              {
                     short lenta_position;      //позиция на ленте
                     RC_STEP rc_step;           //код завершения шага
                     short nrule;               //номер правила
                     short nrule_chain;         //номер цепочки правила
                     MfstDiagnosis();
                     MfstDiagnosis(                     //диагностика
                            short plenta_position,      //позиция на ленте
                            RC_STEP prc_step,           //код завершения шага
                            short pnrule,               //номер правила
                            short pnrule_chain          //номер цепочки правила
                     );
              }
              diagnosis[MFST_DIAGN_NUMBER];     //последние самые глубокие сообщения
              GRBALPHABET* lenta;               //перекодированная(TS/NS) лента (из LEX)
              short lenta_position;             //текущая позиция на ленте
              short nrule;                      //номер текущего правила
              short nrulechain;                 //номер текущей цепочки, текущего правила
              short lenta_size;                 //размер ленты
              GRB::Greibach grebach;            //грамматика Грейбах
              Lexer::LEX lex;                   //результат работы лексического анализатора
              MFSTSTSTACK st;                   //стек автомата
              std::stack<MfstState> storestate; //стек для сохранения состояний
              Mfst();
              Mfst(
                     Lexer::LEX plex,           //результат работы лексического анализатора
                     GRB::Greibach pgrebach     //грамматика Грейбах
              );
              char* getCSt(char*buf);           //получить содержимое стека
              char* getCLenta(char* buf, short pos, short n = 25);//лента: n символов с pos
              char* getDiagnosis(short n, char*buf);//получить n-ую строку диагностики или 0x00
              bool savestate(const Log::LOG &log);//сохранить состояние автомата
              bool reststate(const Log::LOG &log);//восстановить состояние автомата
              bool push_chain(                  //поместить уепочку правила в стек
                     GRB::Rule::Chain chain     //цепочка правила
              );
              RC_STEP step(const Log::LOG &log);//выполнить шаг автомата
              bool start(const Log::LOG &log);  //запустить автомат
              bool savediagnois(
                     RC_STEP pprc_step);        //код завершения шага
              void printrules(const Log::LOG &log);//вывести последовательность правил
              struct Deducation          //вывод
              {
                     short size;         //количество шагов в выводе
                     short* nrules;      //номера правил нрамматики
                     short* nrulechains;//номера цепочек правил грамматики (nrules)
                     Deducation() { size = 0; nrules = 0; nrulechains = 0; };
              } deducation;
              bool savededucation();};    //сохранить дерево вывода
```

Листинг 3 Структура грамматики Грейбах

```
struct Greibach                    //грамматика Грейбах
      {
            short size;                 //количество правил
            GRBALPHABET startN;  //стартовый символ
            GRBALPHABET stbottomT;//дно стека
            Rule *rules;                //множество правил
            Greibach() { short size = 0; startN = 0; stbottomT = 0; rules = 0; };
            Greibach(
                  GRBALPHABET pstartN,       //стартовый символ
                  GRBALPHABET pstbootomT,          //дно стека
                  short psize,                     //количество правил
                  Rule r, ...                            //правила
            );
            short getRule(              //получить правило, возвращается номер правила или -1
                  GRBALPHABET pnn,     //левый символ правила
                  Rule& prule                //возвращаемое правило грамматики
            );
            Rule getRule(short n); };  //получить правило по номеру
```

Листинг 4 Разбор исходного кода синтаксическим анализатором

```
Шаг :Правило          Входная лента          Стек
0   :S->tfiPTS        tfi(ti,ti)[nti;?i<i#w[i=i    S$
1   : SAVESTATE:        1
1   :                 tfi(ti,ti)[nti;?i<i#w[i=i    tfiPTS$
2   :                 fi(ti,ti)[nti;?i<i#w[i=i;    fiPTS$
3   :                 i(ti,ti)[nti;?i<i#w[i=i;]    iPTS$
4   :                 (ti,ti)[nti;?i<i#w[i=i;]r    PTS$
5   :P->(E)           (ti,ti)[nti;?i<i#w[i=i;]r    PTS$
6   : SAVESTATE:        2
6   :                 (ti,ti)[nti;?i<i#w[i=i;]r    (E)TS$
7   :                 ti,ti)[nti;?i<i#w[i=i;]r[    E)TS$
8   :E->ti,E          ti,ti)[nti;?i<i#w[i=i;]r[    E)TS$
9   : SAVESTATE:        3
9   :                 ti,ti)[nti;?i<i#w[i=i;]r[    ti,E)TS$
10  :                  i,ti)[nti;?i<i#w[i=i;]r[i    i,E)TS$
11  :                  ,ti)[nti;?i<i#w[i=i;]r[i=    ,E)TS$
12  :                  ti)[nti;?i<i#w[i=i;]r[i=i    E)TS$
13  :E->ti,E           ti)[nti;?i<i#w[i=i;]r[i=i    E)TS$
14  : SAVESTATE:        4
14  :                  ti)[nti;?i<i#w[i=i;]r[i=i    ti,E)TS$
15  :                  i)[nti;?i<i#w[i=i;]r[i=i;    i,E)TS$
16  :                  )[nti;?i<i#w[i=i;]r[i=i;]    ,E)TS$
17  : TS_NOK/NS_NORULECHAIN
17  : RESSTATE
```

Листинг 4 (прод.) Разбор исходного кода синтаксическим анализатором

```
890 : SAVESTATE:        81
890 :               i(i,i);]                iF;]$
891 :               (i,i);]                 F;]$
892 :F->(N)           (i,i);]                   F;]$
893 : SAVESTATE:        82
893 :               (i,i);]                 (N);]$
894 :               i,i);]                  N);]$
895 :N->i             i,i);]                    N);]$
896 : SAVESTATE:        83
896 :               i,i);]                  i);]$
897 :               ,i);]                   );]$
898 : TS_NOK/NS_NORULECHAIN
898 : RESSTATE
898 :               i,i);]                  N);]$
899 :N->i,N           i,i);]                    N);]$
900 : SAVESTATE:        83
900 :               i,i);]                  i,N);]$
901 :               ,i);]                   ,N);]$
902 :               i);]                    N);]$
903 :N->i             i);]                      N);]$
904 : SAVESTATE:        84
904 :               i);]                    i);]$
905 :               );]                     );]$
906 :               ;]                      ;]$
907 :               ]                       ]$
908 :                                       $
909 : LENTA_END
910 : ------>LENTA_END
```

Приложение Г

Листинг 1 Программная реализация механизма преобразования в ПОЛИЗ

```
bool setPolishNotation(IT::IdTable& idtable, Log::LOG& log, int lextable_pos, ltvec& v)
                        {
                                vector < LT::Entry > result; //результирующий вектор
                                stack < LT::Entry > s; // стек для сохранения операторов
                                bool ignore = false; // флаг вызова функции
                                for (unsigned i = 0; i < v.size(); i++)
                                {
                                        if (ignore)   // вызов функции считаем подставляемым значением
и заносим в результат
                                        {
                                                result.push_back(v[i]);
                                                if (v[i].lexema == LEX_RIGHTTHESIS)
                                                        ignore = false;
                                                continue;
                                        }
                                        int priority = getPriority(v[i]); // его приоритет
                                        if (v[i].lexema == LEX_LEFTHESIS || v[i].lexema ==
LEX_RIGHTTHESIS || v[i].lexema == LEX_PLUS || v[i].lexema == LEX_MINUS || v[i].lexema ==
LEX_STAR || v[i].lexema == LEX_DIRSLASH)
                                        {
                                                if (s.empty() || v[i].lexema == LEX_LEFTHESIS)
                                                        {s.push(v[i]);
                                                        continue;}
                                                if (v[i].lexema == LEX_RIGHTTHESIS)
                                                {//выталкивание элементов до  скобки
                                                while (!s.empty() && s.top().lexema != LEX_LEFTHESIS)
                                                                {result.push_back(s.top());
                                                                s.pop();}
                                                if (!s.empty() && s.top().lexema == LEX_LEFTHESIS)
                                                                s.pop();
                                                        continue;
                                                }
                                                //выталкивание элем с большим/равным приоритетом
                                                while (!s.empty() && getPriority(s.top()) >= priority)
                                                {
                                                        result.push_back(s.top());
                                                        s.pop();
                                                }
                                                s.push(v[i]);
                                        }

                                        if (v[i].lexema == LEX_LITERAL|| v[i].lexema == LEX_ID) //
идентификатор, идентификатор функции или литерал
                                        {
                                                if (idtable.table[v[i].idxTI].idtype == IT::IDTYPE::F
|| idtable.table[v[i].idxTI].idtype == IT::IDTYPE::S)
                                                        ignore = true;
                                                result.push_back(v[i]);// операнд заносим в вектор
                                        }
                                        if (v[i].lexema != LEX_LEFTHESIS & v[i].lexema !=
LEX_RIGHTTHESIS & v[i].lexema != LEX_PLUS & v[i].lexema != LEX_MINUS & v[i].lexema != LEX_STAR &
v[i].lexema != LEX_DIRSLASH &v[i].lexema != LEX_ID & v[i].lexema != LEX_LITERAL)
                                        {
                                                Log::writeError(log.stream, Error::GetError(1));
                                                return false;
                                        }
                                }
                                while (!s.empty()) { result.push_back(s.top()); s.pop(); }
                                v = result;
                                return true;}
```

Листинг 2 Таблица идентификаторов после преобразования выражений в ПОЛИЗ

```
|  N  |СТРОКА В ТЛ| ТИП ИДЕНТИФИКАТОРА |       ИМЯ       | ЗНАЧЕНИЕ (ПАРАМЕТРЫ)
|0    |  2        |  number   function |  min             | P0:NUMBER | P1:NUMBER |
|1    |  5        |  number  parameter |  minx            |
|2    |  8        |  number  parameter |  miny            |
|3    |  13       |  number   variable |  minres          |0
|4    |  41       |   proc    function |  stand           | P0:STRING | P1:STRING |
|5    |  44       | string   parameter |  standa          |
|6    |  47       | string   parameter |  standb          |
|7    |  52       |  number   variable |  standk          |0
|8    |  56       |  number   LIB FUNC |  lenght          | P0:STRING |
|9    |  68       | string    variable |  standstr        |[0]
|10   |  72       | string    LIB FUNC |  concat          | P0:STRING | P1:STRING |
|11   |  91       |  number   variable |  mainx           |0
|12   |  93       |  number    literal |  LTRL1           |9
|13   |  97       |  number   variable |  mainy           |0
|14   |  99       |  number    literal |  LTRL2           |-9
|15   |  103      | string    variable |  mainstrx        |[0]
|16   |  105      | string     literal |  LTRL3           |[4]Just
|17   |  109      | string    variable |  mainstry        |[0]
|18   |  111      | string     literal |  LTRL4           |[6]string
|19   |  115      |  number   variable |  maint           |0
|20   |  133      |  number   variable |  mainab          |0
|21   |  135      |  number    literal |  LTRL5           |3
|22   |  140      |  number    literal |  LTRL6           |52
|23   |  148      | string     literal |  LTRL7           |[1]
|24   |  153      |  number    literal |  LTRL8           |2
```

Листинг 3 Таблица лексем после преобразования выражений в ПОЛИЗ

```
|  N | ЛЕКСЕМА | СТРОКА | ИНДЕКС В ТИ |
|0   | t       |  1     |             |
|1   | f       |  1     |             |
|2   | i       |  1     |0            |
|3   | (       |  1     |             |
|4   | t       |  1     |             |
|5   | i       |  1     |1            |
|6   | ,       |  1     |             |
|7   | t       |  1     |             |
|8   | i       |  1     |2            |
|9   | )       |  1     |             |
|10  | [       |  2     |             |
|11  | n       |  3     |             |
|12  | t       |  3     |             |
|13  | i       |  3     |3            |
|14  | ;       |  3     |             |
|15  | ?       |  4     |             |
|16  | i       |  4     |1            |
|17  | <       |  4     |             |
|18  | i       |  4     |2            |
|19  | #       |  4     |             |
|20  | w       |  5     |             |
|21  | [       |  5     |             |
|22  | i       |  5     |3            |
|23  | =       |  5     |             |
|24  | i       |  5     |1            |
|25  | ;       |  5     |             |
|26  | ]       |  5     |             |
|27  | r       |  6     |             |
|28  | [       |  6     |             |
|29  | i       |  6     |3            |
|30  | =       |  6     |             |
|31  | i       |  6     |2            |
|32  | ;       |  6     |             |
|33  | ]       |  6     |             |
|34  | #       |  6     |             |
|35  | e       |  7     |             |
|36  | i       |  7     |3            |
|37  | ;       |  7     |             |
|38  | ]       |  8     |             |
|39  | p       |  9     |             |
|40  | f       |  9     |             |
|41  | i       |  9     |4            |
|42  | (       |  9     |             |
```

43	t	9		
44	i	9	5	
45	,	9		
46	t	9		
47	i	9	6	
48	)	9		
49	[	10		
50	n	11		
51	t	11		
52	i	11	7	
53	;	11		
54	i	12	7	
55	=	12		
56	i	12	8	
57	(	12		
58	i	12	5	
59	)	12		
60	;	12		
61	o	13		
62	i	13	7	
63	;	13		
64	^	14		
65	;	14		
66	n	15		
67	t	15		
68	i	15	9	
69	;	15		
70	i	16	9	
71	=	16		
72	i	16	10	
73	(	16		
74	i	16	5	
75	,	16		
76	i	16	6	
77	)	16		
78	;	16		
79	o	17		
80	i	17	9	
81	;	17		
82	^	18		
83	;	18		
84	e	19		
85	;	19		
86	]	20		
87	m	21		
88	[	22		
89	n	23		
90	t	23		
91	i	23	11	
92	=	23		
93	l	23	12	
94	;	23		
95	n	24		
96	t	24		
97	i	24	13	
98	=	24		
99	l	24	14	
100	;	24		
101	n	25		
102	t	25		
103	i	25	15	
104	=	25		
105	l	25	16	
106	;	25		
107	n	26		
108	t	26		
109	i	26	17	
110	=	26		
111	l	26	18	
112	;	26		
113	n	27		
114	t	27		
115	i	27	19	
116	;	27		
117	i	28	19	
118	=	28		
119	i	28	0	
120	(	28		
121	i	28	11	
122	,	28		
123	i	28	13	
124	)	28		
125	;	28		
126	o	29		
127	i	29	19	
128	;	29		
129	^	30		
130	;	30		
131	n	31		
132	t	31		
133	i	31	20	
134	=	31		
135	l	31	21	
136	;	31		
137	?	32		
138	i	32	20	
139	!	32		
140	l	32	22	
141	#	32		
142	c	33		
143	[	33		
144	o	34		
145	i	34	20	
146	;	34		

```
|147 | o      | 35   |          |
|148 | l      | 35   |23        |
|149 | ;      | 35   |          |
|150 | i      | 36   |20        |
|151 | =      | 36   |          |
|152 | i      | 36   |20        |
|153 | l      | 36   |24        |
|154 | +      | 36   |          |
|155 | l      | 36   |24        |
|156 | *      | 36   |          |
|157 | ;      | 36   |          |
|158 | ]      | 37   |          |
|159 | #      | 37   |          |
|160 | o      | 38   |          |
|161 | l      | 38   |23
|162 | ;      | 38   |          |
|163 | o      | 39   |          |
|164 | i      | 39   |20        |
|165 | ;      | 39   |          |
|166 | ^      | 40   |          |
|167 | ;      | 40   |          |
|168 | i      | 41   |4         |
|169 | (      | 41   |          |
|170 | i      | 41   |15        |
|171 | ,      | 41   |          |
|172 | i      | 41   |17        |
|173 | )      | 41   |          |
|174 | ;      | 41   |          |
|175 | ]      | 42   |


|
```

Листинг 4 Соответствие лексем исходному коду программы

```
1   | tfi[0](ti[1],ti[2])
2   | [
3   | nti[3];
4   | ?i[1]<i[2]#
5   | w[i[3]=i[1];]
6   | r[i[3]=i[2];]#
7   | ei[3];
8   | ]
9   | pfi[4](ti[5],ti[6])
10  | [
11  | nti[7];
12  | i[7]=i[8](i[5])l[9]+;
13  | ol[10];
14  | oi[7];
15  | ^;
16  | nti[11];
17  | i[11]=i[12](i[5],i[6]);
18  | ol[13];
19  | oi[11];
20  | ^;
22  | e;
23  | ]
24  | m
25  | [
26  | nti[14]=l[15];
27  | nti[16]=l[17];
28  | nti[18]=l[19];
29  | nti[20]=l[21];
30  | nti[22]=l[23];
31  | nti[24];
32  | ol[25];
33  | i[24]=i[26](i[22]);
34  | oi[24];
35  | ^;
36  | nti[27];
37  | i[27]=i[14]l[9]{;
38  | ol[28];
39  | oi[27];
40  | ^;
41  | nti[29];
42  | i[29]=i[0](i[14],i[16]);
43  | oi[29];
44  | ^;
45  | nti[30]=l[31];
46  | nti[32]=q;
47  | ?i[30]!l[33]#
48  | c[
49  | oi[30];
50  | ol[34];
51  | i[30]=i[30]l[35]+l[35]*;
52  | ]#
53  | ol[34];
54  | oi[30];
55  | ^;
56  | i[4](i[18],i[20]);
57  | ]
```

Приложение Д

Листинг 1 Результат генерации кода контрольного примера в Ассемблере

```
.586
.model flat, stdcall
includelib libucrt.lib
includelib kernel32.lib
includelib "../Debug/GenLib.lib
ExitProcess PROTO:DWORD
.stack 4096


 outnum PROTO : DWORD

 outstr PROTO : DWORD

 concat PROTO : DWORD, : DWORD, : DWORD

 lenght PROTO : DWORD, : DWORD

 atoii  PROTO : DWORD

.const
                newline byte 13, 10, 0
                LTRL1 sdword 1
                LTRL2 byte 'Len + 1:', 0
                LTRL3 byte 'concat:', 0
                LTRL4 sdword 9
                LTRL5 sdword -9
                LTRL6 byte 'Just', 0
                LTRL7 byte 'string', 0
                LTRL8 byte '70', 0

        LTRL9 byte 'from string in number:',
 0
                LTRL10 byte 'sdvig left:', 0
                LTRL11 sdword 3
                LTRL12 sdword 52
                LTRL13 byte ' ', 0
                LTRL14 sdword 2
.data
                temp sdword ?
                buffer byte 256 dup(0)
                minres sdword 0
                standk sdword 0
                standstr dword ?
                mainx sdword 0
                mainy sdword 0
                mainstrx dword ?
                mainstry dword ?
                mainstrz dword ?
                maine sdword 0
                mainresult sdword 0
                maint sdword 0
                mainab sdword 0
                maind sdword 0
.code

;----------- min ------------
min PROC,
        minx : sdword, miny : sdword
; --- save registers ---
push ebx
push edx
; ----------------------
mov edx, minx
cmp edx, miny

jl right1
jg wrong1
right1:
push minx

pop ebx
mov minres, ebx

jmp next1
wrong1:
push miny

pop ebx
mov minres, ebx

next1:
; --- restore registers ---
pop edx
pop ebx
; -------------------------
mov eax, minres
ret
min ENDP
;------------------------------


;----------- stand ------------
stand PROC,
        standa : dword, standb : dword
; --- save registers ---
push ebx
push edx
; ----------------------

push standa
push offset buffer
call lenght
push eax
push LTRL1
pop ebx
pop eax
add eax, ebx
push eax

pop ebx
mov standk, ebx


push offset LTRL2
```

```
call outstr


push standk
call outnum

push offset newline
call outstr


push standb
push standa
push offset buffer
call concat
mov standstr, eax

push offset LTRL3
call outstr


push standstr
call outstr

push offset newline
call outstr

; --- restore registers ---
pop edx
pop ebx
; -------------------------
ret
stand ENDP
;------------------------------


;----------- MAIN ------------
main PROC
push LTRL4

pop ebx
mov mainx, ebx

push LTRL5

pop ebx
mov mainy, ebx

mov mainstrx, offset LTRL6
mov mainstry, offset LTRL7
mov mainstrz, offset LTRL8

push offset LTRL9
call outstr


push mainstrz
push offset buffer
call atoii
push eax
```

```
pop ebx
mov maine, ebx


push maine
call outnum

push offset newline
call outstr

push mainx
push LTRL1
pop ebx
pop eax
mov cl, bl
shl eax, cl
push eax

pop ebx
mov mainresult, ebx


push offset LTRL10
call outstr


push mainresult
call outnum

push offset newline
call outstr


push mainy
push mainx
call min
push eax

pop ebx
mov maint, ebx


push maint
call outnum

push offset newline
call outstr

push LTRL11

pop ebx
mov mainab, ebx


pop ebx
mov maind, ebx

mov edx, mainab
cmp edx, LTRL12
```

```
jnz cycle2
jmp cyclenext2
cycle2:

push mainab
call outnum


push offset LTRL13
call outstr

push mainab
push LTRL14
pop ebx
pop eax
add eax, ebx
push eax
push LTRL14
pop ebx
pop eax
imul eax, ebx
push eax

pop ebx
mov mainab, ebx

mov edx, mainab
cmp edx, LTRL12

jnz cycle2
cyclenext2:

push offset LTRL13
call outstr


push mainab
call outnum

push offset newline
call outstr


push mainstry
push mainstrx
call stand

push 0
call ExitProcess
main ENDP
end main
```