МИНИСТЕРСТВО ОБРАЗОВАНИЯ РЕСПУБЛИКИ БЕЛАРУСЬ

Учреждение образования «БЕЛОРУССКИЙ ГОСУДАРСТВЕННЫЙ ТЕХНОЛОГИЧЕСКИЙ УНИВЕРСИТЕТ»

Факультет Информационных Технологий
Кафедра Программной инженерии
Специальность 1-40 01 01 Программное обеспечение информационных технологий

ПОЯСНИТЕЛЬНАЯ ЗАПИСКА К КУРСОВОМУ ПРОЕКТУ НА ТЕМУ:

«Разработка компилятора LMM-2023»

Выполнил студент Ляшонок Матвей Михайлович
(Ф.И.О.)

Руководитель проекта преп.-стажер Север Александра Сергеевна
(учен. степень, звание, должность, подпись, Ф.И.О.)

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

Консультант преп.-стажер Север Александра Сергеевна
(учен. степень, звание, должность, подпись, Ф.И.О.)

Курсовой проект защищен с оценкой

Минск 2023

Содержание

Введение

В данном курсовом проекте поставлена задача разработки собственного языка программирования и транслятора для него. Название языка – LMM-2023. Написание транслятора будет осуществляться на языке C++, при этом код на языке LMM-2023 будет транслироваться в язык ассемблера.

Задание на курсовой проект можно разделить на следующие задачи:

1) Разработка спецификации языка LMM-2023. Для выполнения данной задачи необходимо детально определить лексические единицы - идентификаторы, ключевые слова, оператор присваивания, логические и сравнительные операторы; подробно описать синтаксис – структуру программы на языке, объявление переменных, определение функций, операторы присваивания, ветвления, вывода; изложить спецификацию типов данных – беззнаковые целочисленные; занимаемый объём памяти для каждого типа; правила именования идентификаторов - допустимые символы, регистр, длина имен.

2) Разработка лексического анализатора. С учётом поставленной задачи необходимо разработать алгоритм разбиения исходного кода на лексемы с учетом разделителей, многосимвольных лексем; реализовать распознавание идентификаторов, ключевых слов, обработка ошибок; создать таблицы лексем с их атрибутами (тип, значение, номер строки); механизм выдачи ошибок лексического анализа с указанием номера строк.

3)Разработка синтаксического анализатора. Для достижения целей необходимо рассмотреть алгоритм сопоставления входной последовательности лексем с синтаксическими правилами языка; построение дерева разбора, отражающего иерархическую структуру программы; проверку типов и согласованности в выражениях, присваиваниях, при вызове функций; механизм выдачи ошибок синтаксического анализа.

4) Разработка семантического анализатора. В данной задаче важны: проверка объявлений идентификаторов; анализ присваиваний и вызовов функций, построение графа потока управления, выявление неиспользуемых переменных, выдача предупреждений о потенциальных ошибках;

5)Разбор арифметических выражений. В данной ситуации требуется реализовать алгоритм разбора с учетом приоритетов операций, проверка типов операндов, вычисление значений констант, выдача ошибок и предупреждений, оптимизация выражений.

6)Разработка генератора кода. В контексте задачи важно учесть оптимизацию внутреннего представления программы, распределение регистров.

7)Тестирование транслятора. Для достижения целей стоит протестировать разработанные тестовые программы, выявить и исправить ошибки; использовать регрессионное тестирование; протестировать скомпилированные программы.

1 Спецификация языка программирования

1.1 Характеристика языка программирования

Язык программирования LMM-2023 является статическим сильно типизированным, функциональный, компилируемым.

1.2 Определение алфавита языка программирования

Алфавит языка программирования LMM-2023 использует таблицу символом Windows-1251, представленную ниже на рис 1.1.

	00	01	02	03	04	05	06	07	08	09	0A	0B	0C	0D	0E	0F
00	NUL 0000	STX 0001	SOT 0002	ETX 0003	EOT 0004	ENQ 0005	ACK 0006	BEL 0007	BS 0008	HT 0009	LF 000A	VT 000B	FF 000C	CR 000D	SO 000E	SI 000F
10	DLE 0010	DC1 0011	DC2 0012	DC3 0013	DC4 0014	NAK 0015	SYN 0016	ETB 0017	CAN 0018	EM 0019	SUB 001A	ESC 001B	FS 001C	GS 001D	RS 001E	US 001F
20	SP 0020	! 0021	" 0022	# 0023	$ 0024	% 0025	& 0026	' 0027	(0028	) 0029	* 002A	+ 002B	, 002C	- 002D	. 002E	/ 002F
30	0 0030	1 0031	2 0032	3 0033	4 0034	5 0035	6 0036	7 0037	8 0038	9 0039	: 003A	; 003B	< 003C	= 003D	> 003E	? 003F
40	@ 0040	A 0041	B 0042	C 0043	D 0044	E 0045	F 0046	G 0047	H 0048	I 0049	J 004A	K 004B	L 004C	M 004D	N 004E	O 004F
50	P 0050	Q 0051	R 0052	S 0053	T 0054	U 0055	V 0056	W 0057	X 0058	Y 0059	Z 005A	[005B	\ 005C	] 005D	^ 005E	_ 005F
60	` 0060	a 0061	b 0062	c 0063	d 0064	e 0065	f 0066	g 0067	h 0068	i 0069	j 006A	k 006B	l 006C	m 006D	n 006E	o 006F
70	p 0070	q 0071	r 0072	s 0073	t 0074	u 0075	v 0076	w 0077	x 0078	y 0079	z 007A	{ 007B	| 007C	} 007D	~ 007E	DEL 007F
80	Ъ 0402	Ѓ 0403	‚ 201A	ѓ 0453	„ 201E	… 2026	† 2020	‡ 2021	€ 20AC	‰ 2030	Љ 0409	‹ 2039	Њ 040A	Ќ 040C	Ћ 040B	Џ 040F
90	ђ 0452	‘ 2018	’ 2019	“ 201C	” 201D	• 2022	– 2013	— 2014		™ 2122	љ 0459	› 203A	њ 045A	ќ 045C	ћ 045B	џ 045F
A0	NBSP 00A0	Ў 040E	ў 045E	Ј 0408	¤ 00A4	Ґ 0490	¦ 00A6	§ 00A7	Ё 0401	© 00A9	Є 0404	« 00AB	¬ 00AC	- 00AD	® 00AE	Ї 0407
B0	° 00B0	± 00B1	І 0406	і 0456	ґ 0491	µ 00B5	¶ 00B6	· 00B7	ё 0451	№ 2116	є 0454	» 00BB	ј 0458	Ѕ 0405	ѕ 0455	ї 0457
C0	А 0410	Б 0411	В 0412	Г 0413	Д 0414	Е 0415	Ж 0416	З 0417	И 0418	Й 0419	К 041A	Л 041B	М 041C	Н 041D	О 041E	П 041F
D0	Р 0420	С 0421	Т 0422	У 0423	Ф 0424	Х 0425	Ц 0426	Ч 0427	Ш 0428	Щ 0429	Ъ 042A	Ы 042B	Ь 042C	Э 042D	Ю 042E	Я 042F
E0	а 0430	б 0431	в 0432	г 0433	д 0434	е 0435	ж 0436	з 0437	и 0438	й 0439	к 043A	л 043B	м 043C	н 043D	о 043E	п 043F
F0	р 0440	с 0441	т 0442	у 0443	ф 0444	х 0445	ц 0446	ч 0447	ш 0448	щ 0449	ъ 044A	ы 044B	ь 044C	э 044D	ю 044E	я 044F

Рисунок 1.1 Алфавит входных символов

В исходном коде на языке LMM-2023 допускается использование символов латиницы и кириллицы, цифр от 0 до 9, которые можно употреблять исключительно внутри строковых литералов.

1.3 Применение сепараторов

Специальные символы-разделители используются для разбиения исходного кода программы на лексемы (токены) в процессе лексического анализа. Эти символы-сепараторы выступают в роли границ между отдельными лексическими единицами языка. Список таких разделительных символов приведен в таблице 1.1.

Таблица 1.1 – Сепараторы

Символ(ы)	Назначение
«пробел»	Разделитель цепочек. Допускается везде кроме названий идентификаторов и ключевых слов
{…}	Блок функции или условной конструкции
(…)	Блок фактических или формальных параметров функции, а также приоритет арифметических операций
,	Разделитель параметров функций
A O I	Логические операции
;	Разделитель программных конструкций
=	Оператор присваивания
< > ~ !	Операторы сравнения

Они позволяют отделять идентификаторы, ключевые слова, константы и прочие лексемы друг от друга при разборе исходного кода, чтобы корректно выполнить лексический анализ.

1.4 Применяемые кодировки

В языке программирования для записи исходного кода программ применяется кодировка Windows-1251. Она включает в себя как латинский, так и кириллический алфавиты, а также набор специальных символов. Среди них скобки, круглые скобки ( ), запятая ,, точка с запятой ;, двоеточие :, решетка #, знаки арифметических операций +, -, /, *, знаки сравнения >, <, амперсанд &, восклицательный знак !, ~, а также фигурные скобки {}.

1.5 Типы данных

Характеристика реализованных типов данных языка LMM-2023 представлена в таблице 1.2.

Таблица 1.2 Типы данных языка LMM-2023

Тип данных	Характеристики
Беззнаковый целочисленный тип данных **PosInt**	Фундаментальный тип данных. Используется для работы с числовыми значениями от 0 до 4 294 967 295. При переполнении данного типа будет выведена ошибка. В памяти занимает 4 байта.

Окончание таблицы 1.2

Тип данных	Характеристики
	Поддерживаемые операции: = (бинарный) – оператор присваивания; A (бинарный) – операция побитового и; O (бинарный) – операция побитового или; I (унарный) – операция побитового инвертирования. В качестве условия условного оператора поддерживаются следующие логические операции: < (бинарный) – оператор «меньше»; > (бинарный) – оператор «больше»; ! (бинарный) – оператор неравенства; ~ (бинарный) – оператор равенства;
Символьный тип данных **Litara**	Фундаментальный тип данных. Используется для работы с символом. По умолчанию инициализируется нулевым символом ‘\0’. Занимает 1 байт. Максимальное количество символов – 1. При переполнении данного типа данных будет выведена ошибка. Операции над данными строкового типа: присваивание строковому идентификатору значения другого строкового идентификатора, строкового литерала или значения строковой функции.
Логический тип данных **boolean**	Фундаментальный тип данных (2 байта), используемый для объявления логической переменной, которая принимает одно из двух значений: TRUE или FALSE. Без явно указанной инициализации переменной, присваивается нулевое значение (FALSE).
Строковый тип данных **string**	Фундаментальный тип данных. Предусмотрен для объявления строк. (1 символ – 1 байт). Автоматическая инициализация строкой нулевой длины. Максимальное количество символов в строке – 255.

Язык ориентирован на работу с разными типами данных, позволяет гибко работать с различными типами данных и проводить вычисления над ними в рамках одной программы. Он позволяет работать с беззнаковыми целочисленными значениями, логическими значениями, символами, строками, а также проводить операции над ними.

1.6 Преобразование типов данных

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

1.7 Идентификаторы

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

- состоит из символов латинского алфавита [A-Z|a-z]$^{+}$, кроме символов A, O, I.
- максимальная длина идентификатора равна 15 и не должна превышать это значение.

Пример правильного идентификатора: *numA*, *numB*. Пример неправильного идентификатора: *PosInt* (совпадает с зарезервированным словом).

1.8 Литералы

В языке программирования LMM-2023 существует только 4 типа литералов: целые, символьные, строковые и логические. Их краткое описание представлено в таблице 1.3.

Таблица 1.3 – Литералы

Литералы	Пояснение
Беззнаковые целочисленные литералы.	Десятичное представление может состоять из чисел [0-9]. Минимальное значение равно 0, максимальное 4 294 967 295. При выходе за пределы будет выведена ошибка. Двоичное представление (первый символ: ^) состоит из чисел 0 или 1. При выходе за пределы будет выведена ошибка.
Символьные литералы	Один символ, заключённый в одинарные кавычки.
Логический литерал	Представляет собой ключевые слова FALSE, TRUE, которые являются 0 и 1 соответственно.
Строковые литералы	Символы, заключённые в " " (двойные кавычки), инициализируются пустой строкой.

В языке LMM-2023 недопустимо использование одинарных и двойных кавычек внутри строковых литералов. Для корректного объявления строк необходимо применять односторонние кавычки, то есть строковый литерал должен начинаться и заканчиваться одним и тем же символом одинарной или двойной кавычки. Вложенность разных типов кавычек языка LMM-2023 не поддерживается.

1.9 Объявление данных

При объявлении переменной указываются ключевое слово *var* тип данных и имя идентификатора, и при этом также разрешается провести инициализацию переменной. Объявление или инициализация переменной происходит внутри блока функции или процедуры. Область видимости ограничена программным блоком.

Пример объявления числового типа с инициализацией:
var PosInt val = 24;
Пример объявления переменной символьного типа с инициализацией:
var Litara char = 'Z'.
Для декларации функций важно использовать ключевое слово "method", предварительно указав тип функции. Затем следует обязательное перечисление параметров и определение тела функции, в конце которого возвращается значение. Для декларации процедуры используется ключевое слово "procedure".

1.10 Инициализация данных

При объявлении переменной можно провести начальную инициализацию данных. В этом случае, переменной будет присвоено значение литерала или идентификатора, указанного справа от символа присвоения. При этом объектами-инициализаторами могут быть только идентификаторы или литералы. Если переменная объявляется без явной инициализации, то для типа "PosInt" по умолчанию устанавливается значение 0, для типа Litara – "\0".

1.11 Инструкции языка

При объявлении переменных в языке LMM-2023 предусмотрена возможность их непосредственной инициализации, то есть присваивания начального значения. Применение механизма инициализации позволяет избежать неопределенных значений при последующем использовании переменных. Инструкции языка LMM-2023 представлены в таблице 1.4.

Таблица 1.4 – инструкции языка LMM-2023

Инструкция	Реализация
Объявление переменной	var <тип данных> <идентификатор>;
Объявление переменной с явной инициализацией	var <тип данных> <идентификатор> = <значение>; Значение – инициализатор конкретного типа. Может быть только литералом или идентификатором
Возврат из функции или процедуры	Для функций, возвращающих значение: **back** <идентификатор/литерал>; Возврат значения из процедур не предусмотрен.
Вывод данных	**write** "<идентификатор/литерал>"; **writeline** '<идентификатор/литерал>';

Окончание таблицы 1.4

Инструкция	Реализация
Вызов функции или процедуры	<идентификатор функции> (<список параметров>); Список параметров может быть пустым.
Присваивание	<идентификатор> = <выражение>; Выражением может быть идентификатор, литерал, или вызов функции соответствующего типа. Для беззнакового типа выражение возможно дополнение арифметическими операциями с использованием скобок. Для строкового типа выражение может быть только идентификатором, литералом или вызовом функции, возвращающей значение строкового типа.

В данной таблице рассмотрены базовые инструкции для объявления и инициализации переменных, организации вывода данных, вызова функций и процедур, а также присваивания значений. Отдельное внимание уделено синтаксису инструкций присваивания и возможным видам выражений для разных типов данных.

1.12 Операции языка

В языке LMM-2023 операция умножения имеют более высокий приоритет по сравнению с операциями сложения и вычитания. Так же реализованы побитовые операции и, или, инверсия: операции сравнения. Таблица 1.5 содержит подробное описание доступных операций в языке.

Таблица 1.5 Операции языка LMM-2023

Тип оператора	Оператор
Побитовые	A – побитовое “и” O – побитовое “или” I – инверсия
Присвоения	= –присвоение
Сравнительные	> – больше < – меньше ~ – равенство ! – неравенство
Арифметические	+ – сумма - – разность * – умножение / – деление нацело % – остаток от деления

В таблице описаны побитовые, сравнительные и арифметические операции для работы с данными. Отдельно рассмотрена операция присваивания.

1.13 Выражения и их вычисления

Вычисление выражений представляет собой одну из ключевых задач в языках программирования. Построение любого выражения подчиняется следующим правилам:

1. Для изменения приоритета операций можно использовать скобки.
2. Выражение должно быть записано в одну строку, без переносов.
3. Недопустимо использование двух операторов, следующих друг за другом.
4. В рамках выражения допускается включение вызовов функций, возвращающих беззнаковое целочисленное значение.

Перед генерацией кода каждое выражение преобразуется в обратную польскую запись для упрощения вычисления выражения на языке ассемблера.

1.14 Конструкции языка

Программа на языке LMM-2023 оформляется в виде функций пользователя и главной функции. При составлении функций рекомендуется выделять блоки и фрагменты, представленные в таблице 1.6.

Таблица 1.6 Программные конструкции языка LMM-2023

Конструкция	Реализация
Главная функция	**glavnaya** **{**…**}**
Внешняя функция	<тип данных> **function** <идентификатор> **(**<тип> <идентификатор>, ...**)** **{**… **back** <идентификатор/литерал>; **}**
Внешняя процедура	**procedure** <идентификатор> **(** <тип> <идентификатор>, ...**)** **{**… **back**; **}**
Условная конструкция	**if**(<идентификатор1> <оператор> <идентификатор2>) **блок1{**…**}** **else блок2{**…**}** <идентификатор1>, <идентификатор2> - идентификаторы или литералы беззнакового целочисленного типа. <оператор> - один из операторов сравнения (**> < ! ~**), устанавливающий отношение между двумя операндами. При истинности условия выполняется код внутри блока1, иначе – код внутри блока **блок2**.
Цикл	**cycle** (<условие>) {…}

Язык LMM-2023 имеет 5 основных конструкция, среди которых присутствуют цикл, условная конструкция, процедура.

1.15 Область видимости идентификаторов

Область видимости в языке LMM-2023 организована так же, как в C++, сверху вниз. Это означает, что переменные, объявленные в одной функции, недоступны в других. Все объявления переменных и операции с ними выполняются внутри блоков. Каждая переменная или параметр функции получает префикс, соответствующий имени функции, внутри которой они были объявлены. Все идентификаторы считаются локальными и должны быть объявлены внутри какой-либо функции. Глобальных переменных не существует, и параметры видны только внутри функции, в которой они были объявлены.

1.16 Семантические проверки

В языке программирования LMM-2023 выполняются следующие семантические проверки:

1. Наличие функции **glavnaya** – точки входа в программу;
2. Единственность точки входа;
3. Переопределение идентификаторов;
4. Использование идентификаторов без их объявления;
5. Проверка соответствия типа функции и возвращаемого параметра;
6. Правильность передаваемых в функцию параметров: количество, типы;
7. Правильность символьных выражений;
8. Превышение размера строковых и числовых литералов;
9. Правильность составленного условия условного оператора.

1.17 Распределение оперативной памяти на этапе выполнения

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

1.18 Стандартная библиотека и её состав

В языке LMM-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 LMM-2023

Имя функции	Возвращаемое значение	Принимаемые параметры	Описание
strlength	int	char* str	Функция возвращает длину строки a.
stoi	int	char* str	Функция преобразует строку в число до первого символа не цифры, если преобразование не удается возвращает 0

Окончание таблицы 1.8

strcomp	int	char* str1, char* str2	Функция сравнивает две строки. Возвращает 0 (если равны), 1(первая больше второй), 2(вторая больше первой)
OutputStr	-	char* str	Функция выводит в консоль строку a
OutputLNStr	-	char* str	Функция выводит в консоль строку a, и совершается переход на следующую строку
BREAKL	-	-	Переход на следующую строку
OutputInt	-	unsigned int a	Функция выводит в консоль число a
OutputLNInt	-	unsigned int a	Функция выводит в консоль число a, и совершает переход на следующую строку
OutputChar	-	char a	Функция выводит в консоль символ a
OutputLNChar	-	char a	Функция выводит в консоль символ a, и совершает переход на следующую строку
OutputBool	-	int a	Функция выводит в консоль TRUE(FALSE) в зависимости от параметра a - 1(0)
OutputLNBool	-	int a	Функция выводит в консоль TRUE(FALSE) в зависимости от параметра a - 1(0), и совершает переход на следующую строку

Стандартная библиотека написана на языке C++ и интегрируется с кодом на этапе генерации. Вызовы стандартных функций могут быть осуществлены там, где доступны вызовы пользовательских функций. В стандартной библиотеке также реализованы функции для управления выводом, но они недоступны для конечного пользователя. Для вывода данных предусмотрены операторы “write” и “writeline”.

1.19 Ввод и вывод данных

В языке программирования LMM-2023 ввод данных не поддерживается. Вывод данных осуществляется с помощью оператора **write** или **writeline**. (<идентификатор>|<литерал>);

1.20 Точка входа

В языке LMM-2023 каждая программа должна содержать главную функцию (точку входа) **glavnaya**, с первой инструкции которой начнётся последовательное

выполнение команд программы.

1.21 Препроцессор

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

1.22 Соглашения о вызовах

В языке вызов функций происходит по соглашению о вызовах stdcall.
Особенности stdcall:
– все параметры функции передаются через стек;
– память высвобождает вызываемый код;
– занесение в стек параметров идёт справа налево.

1.23 Объектный код

Язык LMM-2023 транслируется в язык ассемблера.

1.24 Классификация сообщений транслятора

Сообщения, которые генерирует транслятор, влияют на степень информативности транслятора. Следовательно, сообщения от транслятора должны предоставлять наиболее полную информацию о возможных ошибках, допущенных пользователем при написании программы. Вы можете найти список сообщений от транслятора в таблице 1.10.

Таблица 1.10 – Классификация ошибок

Номера ошибок	Характеристика
0 – 110	Системные ошибки
200 – 299	Ошибки лексического анализа
300 – 399	Ошибки семантического анализа
600 – 699	Ошибки синтаксического анализа

Разбиение ошибок на интервалы помогает в разработке логики транслятора путём создания резервных номер, которые будут задействованы в будущем.

1.25 Контрольный пример

Контрольный пример демонстрирует главные особенности языка LMM-2023: его фундаментальные типы, 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 LMM-2022 исходный код транслируется в язык Assembler. Транслятор языка разделён на отдельные части, которые взаимодействуют между собой и выполняют отведённые функции, представленные в пункте 2.1. Получение ассемблерного кода осуществляется с использованием выходных данных лексического анализатора – таблицы лексем и таблицы идентификаторов. Для указания выходных файлов применяются входные параметры транслятора, описанные в таблице 2.1. Структура транслятора языка LMM-2022 представлена на рисунке 1.

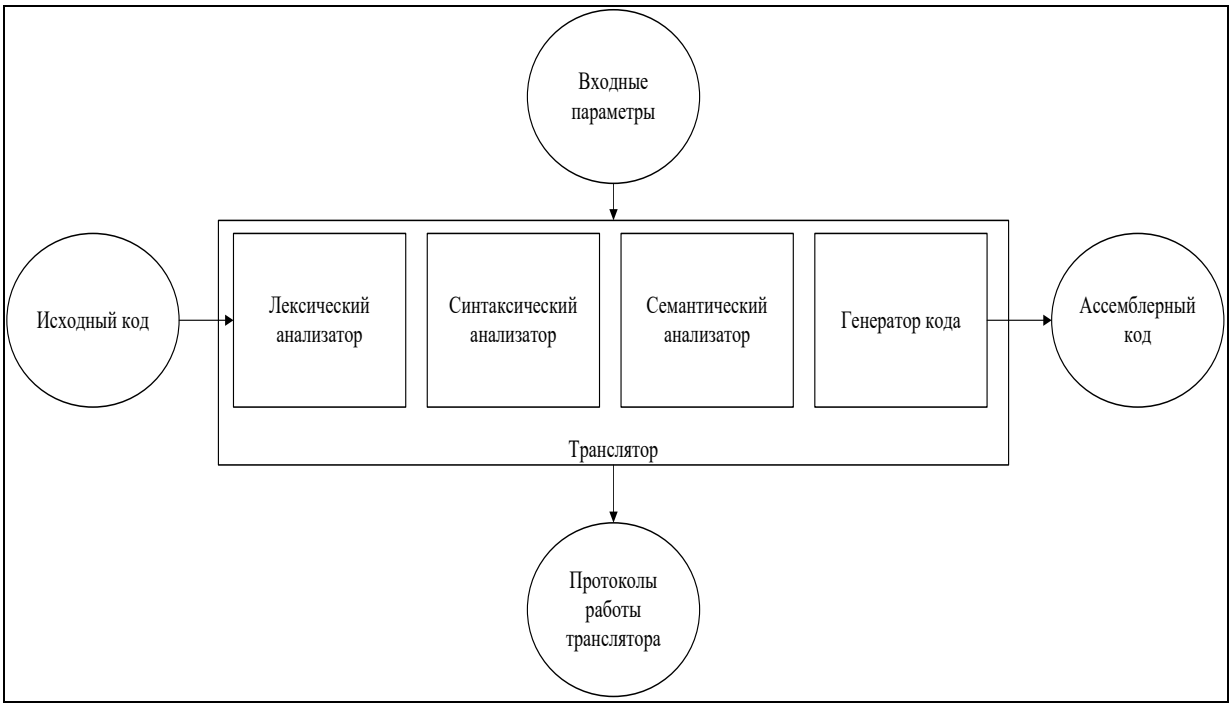

Рисунок 2.1 – Структура транслятора

Лексический анализатор выявляет лексические ошибки в коде LMM-2023 и формирует таблицы лексем и идентификаторов.

Синтаксический анализатор распознает конструкции языка, обнаруживает синтаксические ошибки и строит дерево разбора.

Семантический анализ проверяет правильность программы с точки зрения семантики языка. Генератор кода транслирует проанализированную программу в ассемблер.

2.2 Перечень входных параметров транслятора

Входные параметры представлены в таблице 2.1.

Таблица 2.1 – Входные параметры транслятора языка LMM-2023

Входной параметр	Описание	Значение по умолчанию
-in:<имя_файла>	Входной файл с расширением .txt, в котором содержится исходный код на языке LMM-2023. Данный параметр должен быть указан обязательно. В случае если он не будет задан, то выполнение этапа трансляции не начнётся.	Не предусмотрено
-log:<имя_файла>	Файл содержит в себе краткую информацию об исходном коде на языке LMM-2023. В этот файл могут быть выведены таблицы идентификаторов, лексем, а также дерево разбора.	<имя_файла>.log
-out:<имя_файла>	В этот файл будет записан результат трансляции кода на язык assembler	<имя_файла>.asm
-music	С помощью данного параметра происходит добавление аудио-эффектов, которые воспроизводятся в случае успешного или неудачного компилирования кода.	Не предусмотрено

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

2.3 Перечень протоколов, формируемых транслятором и их содержимое

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

Таблица 2.2 – Протоколы, формируемые транслятором языка LMM-2023

Формируемый протокол	Описание протокола
Файл журнала, log	Файл содержит в себе краткую информацию об исходном коде на языке LMM-2023. В этот файл выводится различные ошибки, а также результат работы анализаторов
Файл журнала, logOut	Файл содержит в себе краткую информацию об исходном коде на языке LMM-2023. В этот файл выводится протокол работы анализаторов.
asm	Содержит сгенерированный код на языке Ассемблера.

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

3. Разработка лексического анализатора

3.1 Структура лексического анализатора

Лексический анализатор – часть транслятора, выполняющая лексический анализ. Лексический анализатор принимает обработанный и разбитый на отдельные компоненты исходный код на языке LMM-2023. На выходе формируется таблица лексем и таблица идентификаторов. Структура лексического анализатора представлена на рисунке 3.1.

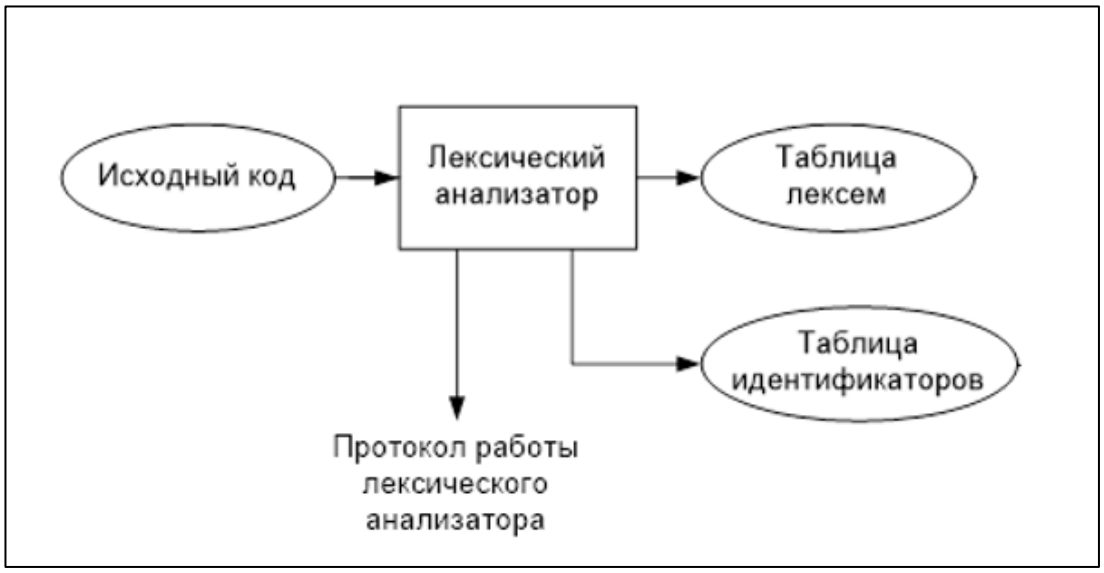

Рисунок 3.1 – Структура лексического анализатора LMM-2023

3.2 Контроль входных символов

Входным параметром лексического анализатора является исходный текст программы, написанный на языке LMM-2023. Таблица для контроля входных символов представлена в листинге 3.1

```
#define IN_CODE_TABLE {\
     IN::T, IN::T, IN::T, IN::T, IN::T, IN::T, IN::T, IN::T, IN::T,
IN::T, IN::BR, 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::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::CMIT, \
     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::I,
     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::T, IN::T, IN::T, IN::T,
IN::T, IN::T, IN::T, IN::T, IN::T, IN::T, 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::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::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::T,
IN::T, IN::T, IN::T, IN::T, IN::T, IN::T, IN::T, \

}
```

Листинг 3.1 – Таблица контроля входных символов

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

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

3.3 Удаление избыточных символов

Избыточными символами являются символы табуляции и пробелы.

Избыточные символы удаляются на этапе разбиения исходного кода на лексемы.

Описание алгоритма удаления избыточных символов:

1. Посимвольно считываем исходный код, занесенный в структуру In.
2. Встреча пробела или знака табуляции вне пределов строкового литерала является своего рода встречей символа-сепаратора.
3. В отличие от других символов-сепараторов не записываем в таблицу лексем эти символы, т.е. игнорируем.

3.4 Перечень ключевых слов, сепараторов, символов операций и соответствующих им лексем, регулярных выражений и конечных автоматов

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

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

Тип цепочки	Цепочка	Лексема
Ключевые слова	var	d
	PosInt, string, boolean, Litara	t
	glavnaya	m
	method	f
	procedure	p
	back	r
	write	o
	writeline	q
	cycle	u
	hortab	b
	if	w
	else	!
Иное	Идентификатор	i
	Литерал	l
Сепараторы	;	;
	,	,
	{	{
	}	}
	(	(
	)	)
Операторы	Арифметические (+, -, *, /, %), Сравнения (!,~,<,>)	v
	Побитовые (A, O,)	v
	Побитовая (I)	n
	Присваивание (=)	=

Пример реализации таблицы лексем представлен в приложении Б.

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

3.5 Основные структуры данных

Описание основных структур данных, используемых для хранения таблиц идентификаторов, представлено на листинге. 3.2.

```
enum IDDATATYPE { NONE = 0, UINT = 1, STR = 2, BOOL = 3, CHAR = 4, PROC = 5 };
enum IDTYPE { V = 1024, F = 2048, P = 2049, L = 2050, OP = 5000 };
struct Parms
{
    IDDATATYPE idDataType; // тип параметра
};
struct Entry // строка таблицы
{
    int idxFirstLine; // индекс первой строки в таблице
    char id[ID_MAXSIZE]; // идентификатор
    IDDATATYPE idDataType; // тиб данных
    IDTYPE idType; // тип
    int countOfPar = 0; // кол-во параметров(функция)
    Parms parm[10]; // параметры
    union
    {
        unsigned int vint; // значение инта
        char vchar[1];
        struct
        {
            unsigned int length;
            char str[TI_STR_MAXSIZE - 1];
        }vstr;// значение стринга

    }value;

};
struct IdTable
{
    int maxSize;
    int size;
    Entry* table;
};
```

Листинг 3.2 – Структуры таблиц идентификаторов LMM-2023

Описание основных структур данных, используемых для хранения таблиц лексем, представлен в листинге. 3.3.

```
struct Entry    //строка таблицы лексем
{
    char lexema;    ///лексема
    int strNumber;          ///№ строки в исх. тексте
    int idxTI;  ///индекс в ТИ или в LT_TI_NULLIDX
    int priority;
    operations op;
    Entry(char lexema, int strNumber, int idxTI)
    {
        this->lexema = lexema;
        this->strNumber = strNumber;
        this->idxTI = idxTI;
    }
    Entry(char lexema, int strNumber)
    {
        this->lexema = lexema;
        this->strNumber = strNumber;
    }
    Entry()
    {

    }
    ~Entry()
    {

    }
};

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

Листинг 3.3 – Структуры таблиц лексем LMM-2023

3.6 Принцип обработки ошибок

Когда возникает ошибка – работа транслятора прекращается, а ошибка записывается в log журнал.

3.7 Структура и перечень сообщений лексического анализатора

Перечень сообщений лексического анализатора представлен на таблице 3.2.

Таблица 3.2 – Сообщения ошибок лексического анализатора

Номер ошибки	Сообщение ошибки
200	Недопустимый символ в исходном файле (-in)
201	Превышен размер таблицы лексем
202	Переполнение таблицы лексем
203	Превышен размер таблицы идентификаторов
204	Переполнение таблицы идентификаторов
205	Неизвестная последовательность символов
207	Неверная запись литерала двоичной системы счисления

Из данной таблицы следует представление об ошибках, которые может определить лексический анализатор.

3.8 Параметры лексического анализатора и режимы его работы

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

3.9 Алгоритм лексического анализа

Лексический анализ выполняется программой (входящей в состав транслятора), называемой лексическим анализатором. Цель лексического анализа — выделение и классификация лексем в тексте исходной программы. Лексический анализатор производит распознаёт и разбирает цепочки исходного текста программы. Это основывается на работе конечных автоматов, которую можно представить в виде графов. Регулярные выражения — аналитический или формульный способ задания регулярных языков. Они состоят из констант и операторов, которые определяют множества строк и множество операций над ними. Любое регулярное выражение можно представить в виде графа. Для ускорения работы анализатора добавлен просмотр первого символа слова, за счет этого отсеиваются неподходящие графы. Результат работы лексического анализатора – сформированные таблицы лексем и идентификаторов.

Пример. Регулярное выражение для ключевого слова glavnaya.

Граф конечного автомата для этой лексемы представлен на рисунке 3.6. S0 – начальное состояние, S8 – конечное состояние автомата.

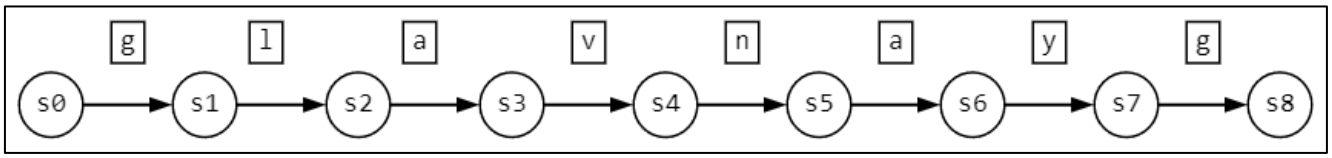

Рисунок 3.6 – Граф переходов для цепочки “glavnaya”

В виде кода граф этой же лексемы представлен в листинге 3.2.

Листинг 3.2 – код регулярного выражения для ключевого слова glavnaya

```
#define FST_MAIN 9, \
     FST::NODE(1, FST::RELATION('g', 1)),\
     FST::NODE(1, FST::RELATION('l', 2)),\
     FST::NODE(1, FST::RELATION('a', 3)),\
     FST::NODE(1, FST::RELATION('v', 4)),\
     FST::NODE(1, FST::RELATION('n', 5)),\
     FST::NODE(1, FST::RELATION('a', 6)),\
     FST::NODE(1, FST::RELATION('y', 7)),\
     FST::NODE(1, FST::RELATION('a', 8)),\
FST::NODE()
```

Показана реализация конечного автомата в виде фрагмента программного кода лексического анализатора. Описан механизм оптимизации за счет предварительного анализа первого символа лексемы.

3.10 Контрольный пример

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

4. Разработка синтаксического анализатора

4.1 Структура синтаксического анализатора

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

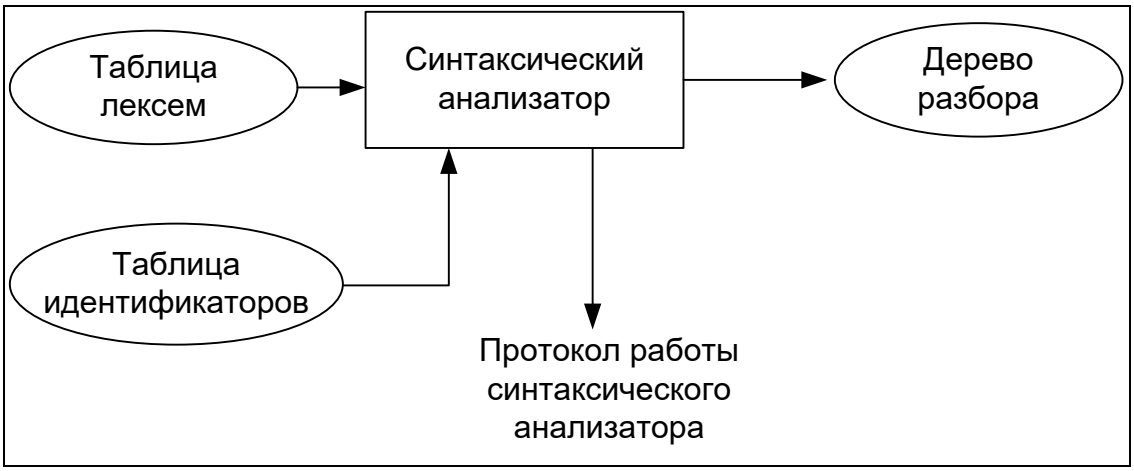

Рисунок 4.1 – Структура синтаксического анализатора

4.2 Контекстно свободная грамматика, описывающая синтаксис языка

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

T – множество терминальных символов

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 Грейбаха, так как 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.

Правила языка LMM-2023 представлена в приложении Г.

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

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

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

Нетерминал	Цепочки правил	Описание
S	ftiFBS piFUS m{N} ftiFB piFU	Проверка правильности структуры программы
F	(P) ()	Проверка наличия списка параметров функции
P	ti ti,P	Проверка на ошибку в параметрах функции при ее объявлении
B	{NrI;} {rI;}	Проверка наличия тела функции
I	l i	Проверка на недопустимое выражение (ожидается только литерал или идентификатор)

N	dti;N
	dti;
	dti=E;N
	dti=E;
	i=E;N
	i=E;
	u(R){L}N
	u(R){ L }
	w(R){ L }N
	w(R){ L }
	w(R){ L }!{ L }N
	w(R){ L }!{ L }
	o[I];N
	o[I];
	b;N
	b;
	iK;N
	iK;
		Проверка на неверную конструкцию в теле функции

K	(W) ()	Проверка на ошибку в вызове функции
E	i iM l lM (E) (E)M iK iKM	Проверка на ошибку в арифметическом выражении

Окончание таблицы 4.1

Нетерминал	Цепочки правил	Описание
W	i l i,W l,W	Проверка на ошибку в параметрах вызываемой функции
M	vE vEM	Проверка арифметических действий
L	i=E;N i=E; oi;N oK; b;N b; iK;N iK;	Проверка на неверную конструкцию в теле цикла или условного выражения

Рассмотрены нетерминальные символы, представляющие различные синтаксические конструкции языка. Приведены правила вывода для каждого нетерминала и дано их описание.

4.3 Построение конечного магазинного автомата

Конечный автомат с магазинной памятью представляет собой семерку $M = \langle Q, V, Z, \delta, q_0, z_0, F \rangle$, описание которой представлено в таблице 4.2. Структура данного автомата показана в приложении Д.

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 является множеством терминальных и нетерминальных символов, описание которых содержится в разделе 1.2 и в таблице 4.1.
Z	Алфавит специальных магазинных символов	Алфавит магазинных символов содержит стартовый символ и маркер дна стека
δ	Функция переходов автомата	Функция представляет из себя множество правил грамматики, описанных в таблице 4.1.

Окончание таблицы 4.2

Компонента	Определение	Описание
q_0	Начальное состояние автомата	Состояние, которое приобретает автомат в начале своей работы. Представляется в виде стартового правила грамматики (нетерминальный символ A)
z_0	Начальное состояние магазина автомата	Символ маркера дна стека ($)
F	Множество конечных состояний	Конечные состояние заставляют автомат прекратить свою работу. Конечным состоянием является пустой магазин автомата и совпадение позиции на входной ленте автомата с размером ленты

4.4 Основные структуры данных

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

4.5 Описание алгоритма синтаксического разбора

Принцип работы автомата следующий:

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

4.6 Структура и перечень сообщений синтаксического анализатора

Перечень сообщений синтаксического анализатора представлен в таблице 4.3.

Таблица 4.3 – Сообщения ошибок синтаксического анализатора

Номер ошибки	Сообщение ошибки
600	Неверная структура программы
601	Неверная конструкция функции
602	Ошибка в выражении

Окончание таблицы 4.3

Номер ошибки	Сообщение ошибки
603	Ошибка объявления функции или процедуры
604	Ошибка в параметрах функции или процедуры
605	Неверная структура метода
606	Неверная структура процедуры
607	Ошибка в параметрах вызываемой функции
608	Неверное условие цикла или условного оператора
609	Ошибка при вызове функции
610	Ошибка в операторе
611	Неверная конструкция цикла или условного оператора
612	Синтаксический анализатор экстренно завершил работу
613	Превышена длина идентификатора

В данной таблице предоставлен перечень ошибок синтаксического анализатора для более подробного ознакомления.

4.7 Параметры синтаксического анализатора и режимы его работы

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

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

4.8 Принцип обработки ошибок

Обработка ошибок происходит следующим образом:

1. Синтаксический анализатор перебирает все правила и цепочки правила грамматики для нахождения подходящего соответствия с конструкцией, представленной в таблице лексем.
2. Если невозможно подобрать подходящую цепочку, то генерируется соответствующая ошибка.
3. Синтаксический анализатор работает до 3 ошибок.
4. В случае нахождения ошибки, после всей процедуры трассировки в протокол будет выведено диагностическое сообщение.

4.9 Контрольный пример

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

5. Разработка семантического анализатора

5.1 Структура семантического анализатора

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

5.2 Функции семантического анализатора

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

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

5.3 Структура и перечень сообщений семантического анализатора

Сообщения, формируемые семантическим анализатором, представлены в таблице 5.1.

Таблица 5.1 – Сообщения ошибок семантического анализатора

Номер ошибки	Сообщение ошибки
300	Имеется незакрытый строковый литерал
301	Имеется более одной точки входа в glavnaya
302	Не имеется точки входа в glavnaya
303	Превышен размер символьного литерала
304	Недопустимый тип данных для сравнения
305	Необъявленный идентификатор
306	Попытка переопределить переменную
307	Попытка реализовать существующую функцию
308	Имеется незакрытый символьный литерал
309	Несовпадение фактических и формальных параметров функции
310	Недопустимый оператор в условии
311	Недопустим вывод арифметического выражения или вызова функции
312	Несоответствие типов данных
313	Несоответствие открытых и закрытых скобок в выражении
314	Функция возвращает неверный тип данных
315	Несовпадение типов данных в условном операторе или цикле
316	Тип данных в условном операторе не является boolean
317	Значение переменной типа string выходит за пределы допустимых значений

Окончание таблицы 5.1

Номер ошибки	Сообщение ошибки
318	Превышение значения переменной допустимого размера типа данных

Таблица представляет перечень ошибок формируемые семантическим анализатором.

5.4 Принцип обработки ошибок

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

Обработка ошибок происходит следующим образом:

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

5.5 Контрольный пример

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

Таблица 5.1 – Тестирование функций

Исходный код с ошибкой	Генерируемое сообщение об ошибке

{
var PosInt check = 5;
write check;
}
	ERROR CODE 302: { SEMANTIC } Не имеется точки входа в glavnaya
	Строка -1 позиция -1

glavnaya
{
var PosInt check = 5;
write check;
write "hello;
}
	ERROR CODE 300: { SEMANTIC } Имеется незакрытый строковый литерал
	Строка -1 позиция -1

glavnaya
{
var PosInt check = 5;
var PosInt check = 2;
}
	ERROR CODE 306, { SEMANTIC } Попытка переопределить переменную/функцию
	Строка n позиция -1

6. Вычисление выражений

6.1 Выражения, допускаемые языком

В языке LMM-2023 допускаются выражения, применимые к целочисленным типам данных. В выражениях поддерживаются арифметические операции, такие как +, -, *, /, %(остаток от деления), побитовые операции A(и), O(или), I(инверсия),сравнительные операции, (), и вызовы функций как операнды арифметических выражений.

Приоритет операций представлен в таблице 6.1.

Таблица 6.1 – Приоритет операций в языке LMM-2023

Приоритет	Операция
0	( ) < > ! ~
1	,
2	+ -
3	* / %
4	A I O

Операции в языке LMM-2023 имеют различный приоритет, который определяет порядок вычисления операндов в выражениях. Такая иерархия приоритетов позволяет однозначно определять порядок вычисления любых арифметических и побитовых выражений

6.2 Польская запись

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

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

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

Алгоритм построения:

– исходная строка: выражение;

– результирующая строка: польская запись;

– стек: пустой;

– результирующая строка: польская запись;

– исходная строка просматривается слева направо;

– операнды переносятся в результирующую строку в порядке их следования;

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

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

– запятая не помещается в стек, в случае если в стеке операции, то все

выбираются в строку;

– отрывающая скобка помещается в стек;

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

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

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

6.3 Программная реализация обработки выражений

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

6.4 Контрольный пример

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

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

Таблица 6.2 – Пример преобразования выражения в обратную польскую запись

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

Таким образом преобразование к польской записи поможет в формировании кода ассемблера на этапе генерации.

7. Генерация кода

7.1 Структура генератора кода

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

Структура генератора кода представлена на рисунке 7.1.

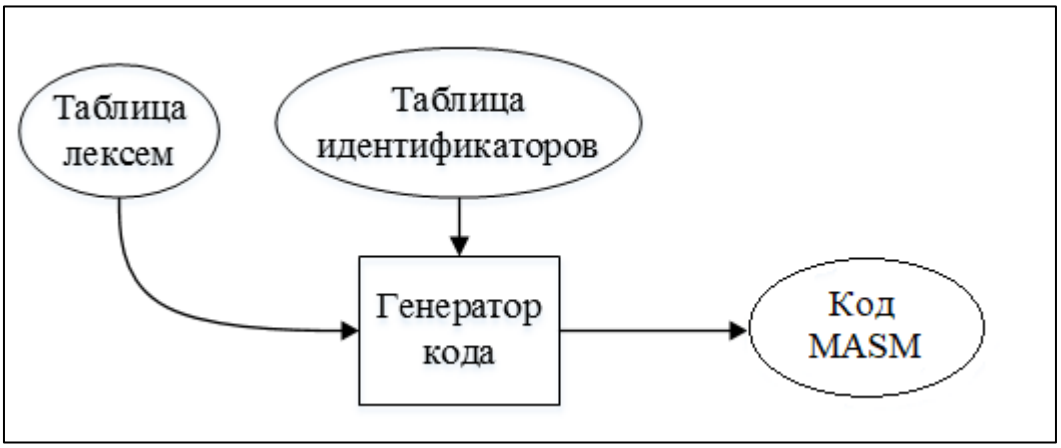

Рисунок 7.1 Структура генератора кода

Функция генератора кода представлена на листинге 7.1.

Листинг 7.1 – код функций генерации кода в язык ассемблера

```
Generator::Generator(LT::LexTable plexT, IT::IdTable pidT, wchar_t
pout[])
{
	lexT = plexT;
	idT = pidT;
	out = std::ofstream(pout, std::ios_base::out);

	Head();
	Const();
	Data();
	Code();
	out.close();
}
```

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

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

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

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

7.2 Представление типов данных в оперативной памяти

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

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

Тип идентификатора на языке LMM-2023	Тип идентификатора на языке ассемблера	Пояснение
PosInt	SDWORD	Хранит целочисленный беззнаковый тип данных со знаком. 4 байта.
string	BYTE	Каждый символ строки хранится размером в 1 байт.
Litara	word	Символ хранится размером в 2 байт.
boolean	word	Переменная хранится размером в 2 байт.

7.3 Статическая библиотека

Функции из стандартной библиотеки содержатся в проекте StaticLibrary, в свойствах которого указан тип конфигурации «статическая библиотека». Подключение библиотеки происходит с помощью includelib на этапе генерации кода путем вывода в поток out. Таким же образом с помощью оператора EXTRN объявляются названия функций из библиотеки. Оператор EXTRN выполняет две функции. Во-первых, он сообщает ассемблеру, что указанное символическое имя является внешним для текущего ассемблирования. Вторая функция оператора EXTRN состоит в том, что он указывает ассемблеру тип соответствующего символического имени. Так как ассемблирование является очень формальной процедурой, то ассемблер должен знать, что представляет из себя каждый символ. Это позволяет ему генерировать правильные команды. Вышеописанное записано в листинг 7.2.

```
void Generator::Head()  // подключение lib и прототипы функций
{
     out << ".586\n";
     out << ".model flat, stdcall\n";
```

```
     out << "includelib libucrt.lib\n";
     out << "includelib kernel32.lib\n";
     out << "includelib D:\\Desktop\\Экзы\\LMM-
2023\\Debug\\StaticLibrary.lib\n";
     out << "ExitProcess proto :dword\n\n";
     out << "EXTRN BREAKL: proc\n";
     out << "EXTRN OutputInt: proc\n";
     out << "EXTRN OutputStr: proc\n";
     out << "EXTRN OutputChar: proc\n";
     out << "EXTRN OutputBool: proc\n";
     out << "EXTRN OutputLNInt: proc\n";
     out << "EXTRN OutputLNStr: proc\n";
     out << "EXTRN OutputLNChar: proc\n";
     out << "EXTRN OutputLNBool: proc\n";
     out << "EXTRN strlength: proc\n";
     out << "EXTRN stoi: proc\n";
     out << "EXTRN strcomp: proc\n";
     out << "\n.stack 4096\n\n";
}
```

Листинг 7.2 – фрагмент генерации кода

Листингом наглядно показана структура подключения функций из стандартной библиотеки проекта StaticLibrary.

8. Тестирование транслятора

8.1 Общие положения

Тестирование транслятора проводится с целью проверки корректности его работы и выявления возможных ошибок. Для тестирования используется набор тестовых программ на языке LMM-2023, включающий как корректные программы, так и программы, содержащие различные ошибки синтаксиса и семантики.

При обнаружении ошибки в исходной программе транслятор должен выдавать сообщение, указывающее строку и краткое описание ошибки. Трансляция при этом не начинается.

Результаты тестирования заносятся в протокол тестирования, который содержит коды завершения транслятора, а также тексты сообщений об ошибках. На основании этого протокола делается вывод о корректности работы транслятора.

8.2 Результаты тестирования

Результаты тестирования представлены в таблице 8.1. Для каждого фрагмента кода с ошибкой представлен код ошибки, соответствующий этапу трансляции, на котором произошло обнаружение ошибки, текст диагностического сообщения транслятора и указание на место ошибки в исходном коде.

Таблица 8.1 – Тестирование фазы проверки на допустимость символов

Исходный код	Диагностическое сообщение

met№od PosInt counts(PosInt k)
{
…
back k;
}
	ERROR CODE 200: { LEXICAL } Недопустимый символ в исходном файле (-in)
	Строка 1 позиция 4

var Litara g = 'f;	ERROR CODE 308: { SEMANTIC } Имеется незакрытый символьный литерал Строка 1 позиция -1

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

В языке LMM-2023 не разрешается использовать запрещённые входным алфавитом символы, проверяется конструкция литералов и функций

Заключение

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

1. Сформулирована спецификация языка LMM-2023;
2. Разработаны конечные автоматы и алгоритмы для эффективной работы лексического анализатора;
3. Осуществлена программная реализация лексического анализатора, распознающего лексемы языка;
4. Разработана контекстно-свободная грамматика языка в нормальной форме Грейбаха;
5. Реализован синтаксический анализатор на основе разработанной грамматики;
6. Разработан семантический анализатор, проверяющий семантику программ;
7. Реализован транслятор в язык ассемблера на основе промежуточного представления программ;
8. Разработан интерпретатор с поддержкой отладочных сообщений;
9. Проведено тестирование всех компонентов транслятора и интерпретатора.

Окончательная версия языка LMM-2023 включает:

1. Целочисленный беззнаковый тип данных, логический, строковый и символьный;
2. Операторы вывода строк, символов, идентификаторов;
3. Вызовы функций стандартной библиотеки;
4. Арифметические, сравнительные, побитовые и логические операции;
5. Поддержку функций, процедур, циклов и ветвлений;
6. Систему диагностики ошибок.

Работа позволила получить опыт разработки компилятора и интерпретатора для собственного языка программирования.

Общее число строк исходного кода – более 4 тысяч.

Приложение А

Исходный код программы на языке LMM-2023

```
procedure Count( string h)
{
  write "счет от нуля до длины строки: ";
  write h;
  hortab;
  var PosInt length = strlength(h);
  var PosInt i= 0;
  cycle (i < length)
  {
   write i;
   write ',';
   i=i+1;
  }
}
PosInt method GetFact( PosInt n)
{
  var PosInt k;
  var PosInt j;
 if (n < 0)
 {
   k = 0;
 }

 if (n ~ 0)
 {
   k = 1;
 }
 else
 {
   j=n- 1;
   k=n*GetFact(j);
 }
back k;
}
glavnaya
{
var PosInt or;
or = 25 O 5;
var PosInt and = 7 A 3;
var PosInt inv = I ^01111111111111111111111111110000 ;
var string vyvod = "побитовые операции: или, и, инверсия:";
writeline vyvod;
write or;
hortab;
writeline and;
writeline inv;
var Litara test = 'n';
writeline test;
writeline "особенности беззнакового целочисленного ципа
данных:";
```

```
var PosInt someN = 5;
writeline someN;
?someN = -1;? ошибка(
someN = someN - 10;
writeline someN;
Count(vyvod);
writeline " результат выполнения процедуры";
var PosInt fac = stoi("3asd");
writeline fac;
fac = GetFact(fac);
writeline fac;
if(someN ! 30)
{
 writeline "someN не равно 30";
}
else
{
write "что?";
}
var PosInt comp = strcomp("world", "words");
}
```

Приложение Б

Таблица идентификаторов

```
ID Table
------------------------------------------------------------------------
------------------
 №   | Identifier    | Data Type  | Identifier Type    | Index in LT
| Value
------------------------------------------------------------------------
------------------
0000 | Count         | void       | процедура          | 1
| -
0001 | Counth        | string     | параметр           | 4
| -
0002 | L1            | string     | литерал            | 8
| [30]"счет от нуля до длины строки: "
0003 | Countlength   | integer    | переменная         | 17
| 0
0004 | strlength     | integer    | метод              | 19
| -
0005 | Counti        | integer    | переменная         | 26
| 0
0006 | L2            | integer    | литерал            | 28
| 0
0007 | L3            | symbol     | литерал            | 41
| ','
0008 | L4            | integer    | литерал            | 47
| 1
0009 | GetFact       | integer    | метод              | 53
| -
0010 | GetFactn      | integer    | параметр           | 56
| -
0011 | GetFactk      | integer    | переменная         | 61
| 0
0012 | GetFactj      | integer    | переменная         | 65
| 0
0013 | glavnaya      | integer    | метод              | 113
| -
0014 | glavnayaor    | integer    | переменная         | 117
| 0
0015 | L5            | integer    | литерал            | 121
| 25
0016 | L6            | integer    | литерал            | 123
| 5
0017 | glavnayaand   | integer    | переменная         | 127
| 0
0018 | L7            | integer    | литерал            | 129
| 7
0019 | L8            | integer    | литерал            | 131
| 3
0020 | glavnayainv   | integer    | переменная         | 135
| 0
0021 | L9            | integer    | литерал            | 138
```

```
| 4294967280
0022 | glavnayavyvod | string     | переменная          | 142
| [0]""
0023 | L10           | string     | литерал             | 144
| [37]"побитовые операции: или, и, инверсия:"
0024 | glavnayatest  | symbol     | переменная          | 162
| ''
0025 | L11           | symbol     | литерал             | 164
| 'n'
0026 | L12           | string     | литерал             | 170
| [52]"особенности беззнакового целочисленного ципа данных:"
0027 | glavnayasomeN | integer    | переменная          | 174
| 0
0028 | L13           | integer    | литерал             | 185
| 10
0029 | L14           | string     | литерал             | 196
| [31]" результат выполнения процедуры"
0030 | glavnayafac   | integer    | переменная          | 200
| 0
0031 | stoi          | integer    | метод               | 202
| -
0032 | L15           | string     | литерал             | 204
| [4]"3asd"
0033 | L16           | integer    | литерал             | 224
| 30
0034 | L17           | string     | литерал             | 228
| [17]"someN не равно 30"
0035 | L18           | string     | литерал             | 234
| [4]"что?"
0036 | glavnayacomp  | integer    | переменная          | 239
| 0
0037 | strcomp       | integer    | метод               | 241
| -
0038 | L19           | string     | литерал             | 243
| [5]"world"
0039 | L20           | string     | литерал             | 245
| [5]"words"
-------------------------------------------------------------------
-------------------
Количество идентификаторов: 40
-------------------------------------------------------------------
-------------------
```

Таблица лексем

```
Lexical Table
----------------------------------------
№    | lex  | idx in IT |номер строки |
----------------------------------------
0000 | p    |           | 1           |
0001 | i    | 0         | 1           |
0002 | (    |           | 1           |
0003 | t    |           | 1           |
```

```
0004 | i    | 1          | 1           |
0005 | )    |            | 1           |
0006 | {    |            | 2           |
0007 | o    |            | 3           |
0008 | l    | 2          | 3           |
0009 | ;    |            | 3           |
0010 | o    |            | 4           |
0011 | i    | 1          | 4           |
0012 | ;    |            | 4           |
0013 | b    |            | 5           |
0014 | ;    |            | 5           |
…
0248 | d    |            | 70          |
0249 | t    |            | 70          |
0250 | i    | 49         | 70          |
0251 | =    |            | 70          |
0252 | l    | 50         | 70          |
0253 | v    | 9          | 70          |
0254 | l    | 50         | 70          |
0255 | v    | 17         | 70          |
0256 | l    | 50         | 70          |
0257 | ;    |            | 70          |
0258 | o    |            | 71          |
0259 | i    | 49         | 71          |
0260 | ;    |            | 71          |
0261 | }    |            | 72          |
```

Промежуточное представление кода

```
0001| pi(ti){ol;
0002| oi;
0003| b;
0004| dti=i(i);
0005| dti=l;
0006| u(ivi){oi;
0007| ol;
0008| i=ivl;
0009| }}tfi(ti){dti;
0010| dti;
0011| w(ivl){i=l;
0012| }w(ivl){i=l;
0013| }!{i=ivl;
0014| i=ivi(i);
0015| }ri;
0016| }m{dti;
…
0037| i=i(i);
0038| qi;
0039| w(ivl){ql;
0040| }!{ol;
0041| }dti=i(l,l);
0042| }
```

Приложение В

Конечные автоматы, соответствующие лексемам языка LMM-2023.

```
#define FST_DECLARE 4,    \
     FST::NODE(1, FST::RELATION('v', 1)),\
     FST::NODE(1, FST::RELATION('a', 2)),\
     FST::NODE(1, FST::RELATION('r', 3)),\
     FST::NODE()

#define FST_INTEGER 7,    \
     FST::NODE(1, FST::RELATION('P', 1)),\
     FST::NODE(1, FST::RELATION('o', 2)),\
     FST::NODE(1, FST::RELATION('s', 3)),\
     FST::NODE(1, FST::RELATION('I', 4)),\
     FST::NODE(1, FST::RELATION('n', 5)),\
     FST::NODE(1, FST::RELATION('t', 6)),\
     FST::NODE()

#define FST_BOOL 8, \
     FST::NODE(1, FST::RELATION('b', 1)),\
     FST::NODE(1, FST::RELATION('o', 2)),\
     FST::NODE(1, FST::RELATION('o', 3)),\
     FST::NODE(1, FST::RELATION('l', 4)),\
     FST::NODE(1, FST::RELATION('e', 5)),\
     FST::NODE(1, FST::RELATION('a', 6)),\
     FST::NODE(1, FST::RELATION('n', 7)),\
     FST::NODE()

#define FST_CHAR 7, \
     FST::NODE(1, FST::RELATION('L', 1)),\
     FST::NODE(1, FST::RELATION('i', 2)),\
     FST::NODE(1, FST::RELATION('t', 3)),\
     FST::NODE(1, FST::RELATION('a', 4)),\
     FST::NODE(1, FST::RELATION('r', 5)),\
     FST::NODE(1, FST::RELATION('a', 6)),\
     FST::NODE()

#define FST_STRING 7,     \
     FST::NODE(1, FST::RELATION('s', 1)),\
     FST::NODE(1, FST::RELATION('t', 2)),\
     FST::NODE(1, FST::RELATION('r', 3)),\
     FST::NODE(1, FST::RELATION('i', 4)),\
     FST::NODE(1, FST::RELATION('n', 5)),\
     FST::NODE(1, FST::RELATION('g', 6)),\
     FST::NODE()

#define FST_FUNCTION 7,  \
     FST::NODE(1, FST::RELATION('m', 1)),\
     FST::NODE(1, FST::RELATION('e', 2)),\
     FST::NODE(1, FST::RELATION('t', 3)),\
     FST::NODE(1, FST::RELATION('h', 4)),\
     FST::NODE(1, FST::RELATION('o', 5)),\
     FST::NODE(1, FST::RELATION('d', 6)),\
```

```
     FST::NODE()
…

#define FST_LEFTBRACE 2, \
     FST::NODE(1, FST::RELATION('{', 1)),\
     FST::NODE()

#define FST_BRACELET 2,  \
     FST::NODE(1, FST::RELATION('}', 1)),\
     FST::NODE()

#define FST_LEFTTHESIS 2,     \
     FST::NODE(1, FST::RELATION('(', 1)),\
     FST::NODE()

#define FST_RIGHTTHESIS 2,    \
     FST::NODE(1, FST::RELATION(')', 1)),\
     FST::NODE()

#define FST_EQUAL 2,     \
     FST::NODE(1, FST::RELATION('=', 1)),\
     FST::NODE()
#define FST_INVERTING 2,\
     FST::NODE(1,FST::RELATION('I',1)),\
     FST::NODE()

#define FST_AND 2,\
     FST::NODE(1,FST::RELATION('A',1)),\
     FST::NODE()
#define FST_OR 2,\
     FST::NODE(1,FST::RELATION('O',1)),\
     FST::NODE()
```

Приложение Г

```
Greibach greibach(

     NS('S'), TS('$'),
     13,
     Rule(
          NS('S'), GRB_ERROR_SERIES + 0,// неверная структура
программы
          5,
          Rule::Chain(6, TS('t'), TS('f'), TS('i'), NS('F'),
NS('B'), NS('S')),
          Rule::Chain(5, TS('p'), TS('i'), NS('F'), NS('U'),
NS('S')),
          Rule::Chain(4, TS('m'), TS('{'), NS('N'), TS('}')),
          Rule::Chain(5, TS('t'), TS('f'), TS('i'), NS('F'),
NS('B')),
          Rule::Chain(4, TS('p'), TS('i'), NS('F'), NS('U'))
     ),
     Rule(
          NS('N'), GRB_ERROR_SERIES + 1, // неверная конструкция
фунцкии
          20,
          Rule::Chain(5, TS('d'), TS('t'), TS('i'), TS(';'),
NS('N')),
          Rule::Chain(7, TS('d'), TS('t'), TS('i'), TS('='),
NS('E'), TS(';'), NS('N')),
          Rule::Chain(5, TS('i'), TS('='), NS('E'), TS(';'),
NS('N')),
          Rule::Chain(8, TS('u'), TS('('), NS('C'), TS(')'),
TS('{'), NS('L'), TS('}'), NS('N')),
          Rule::Chain(8, TS('w'), TS('('), NS('C'), TS(')'),
TS('{'), NS('L'), TS('}'), NS('N')),
          Rule::Chain(12, TS('w'), TS('('), NS('C'), TS(')'),
TS('{'), NS('L'), TS('}'), TS('!'), TS('{'), NS('L'), TS('}'),
NS('N')),
          Rule::Chain(4, TS('o'), NS('I'), TS(';'), NS('N')),
          Rule::Chain(3, TS('o'), NS('I'), TS(';')),
          Rule::Chain(3, TS('q'), NS('I'), TS(';')),
          Rule::Chain(4, TS('q'), NS('I'), TS(';'), NS('N')),



          Rule::Chain(4, TS('i'), NS('K'), TS(';'), NS('N')),
          Rule::Chain(3, TS('b'), TS(';'), NS('N')),
          Rule::Chain(2, TS('b'), TS(';')),
          Rule::Chain(4, TS('d'), TS('t'), TS('i'), TS(';')),
          Rule::Chain(6, TS('d'), TS('t'), TS('i'), TS('='),
NS('E'), TS(';')),
          Rule::Chain(4, TS('i'), TS('='), NS('E'), TS(';')),
          Rule::Chain(7, TS('u'), TS('('), NS('C'), TS(')'),
TS('{'), NS('L'), TS('}')),
```

```
          Rule::Chain(7, TS('w'), TS('('), NS('C'), TS(')'),
TS('{'), NS('L'), TS('}')),
          Rule::Chain(11, TS('w'), TS('('), NS('C'), TS(')'),
TS('{'), NS('L'), TS('}'), TS('!'), TS('{'), NS('L'), TS('}')),

          Rule::Chain(3, TS('i'), NS('K'), TS(';'))
     ),
     Rule(
          NS('E'), GRB_ERROR_SERIES + 2, // ошибка в выражении
          9,
          Rule::Chain(1, TS('i')),
          Rule::Chain(1, TS('l')),
          Rule::Chain(3, TS('('), NS('E'), TS(')')),
          Rule::Chain(2, TS('i'), NS('K')),
          Rule::Chain(2, TS('i'), NS('M')),
          Rule::Chain(2, TS('l'), NS('M')),
          Rule::Chain(4, TS('('), NS('E'), TS(')'), NS('M')),
          Rule::Chain(3, TS('i'), NS('K'), NS('M')),
          Rule::Chain(2, TS('n'), NS('I'))
     ),
     Rule(
          NS('F'), GRB_ERROR_SERIES + 3,// ошибка объявления
функции или процедуры
          2,
          Rule::Chain(3, TS('('), NS('P'), TS(')')),
          Rule::Chain(2, TS('('), TS(')'))
     ),
     Rule(
          NS('P'), GRB_ERROR_SERIES + 4, // ошибка в параметрах
функции или процедуры
          2,
          Rule::Chain(2, TS('t'), TS('i')),
          Rule::Chain(4, TS('t'), TS('i'), TS(','), NS('P'))
     ),
     Rule(
          NS('B'), GRB_ERROR_SERIES + 5, //  неверная структура
метода
          2,
          Rule::Chain(6, TS('{'), NS('N'), TS('r'), NS('I'),
TS(';'), TS('}')),
          Rule::Chain(5, TS('{'), TS('r'), NS('I'), TS(';'),
TS('}'))
     ),
     Rule(
          NS('U'), GRB_ERROR_SERIES + 6, // неверная структура
процедуры
          1,
          Rule::Chain(3, TS('{'), NS('N'), TS('}'))
     ),
```

Приложение Д

Структура конечного автомата

```
struct MfstState // состояние автомата (для сохранения)
{
	short nrule;
	short lentaPosition; // позиция на ленте
	short numRuleChain; // номер текущей цепочки, текущего правила
	MFSTSTSTACK stack;  // стек автомата
	MfstState();
	MfstState
	(
		short position, // позиция на ленте
		MFSTSTSTACK pst, // стек автомата
		short pnRuleChain // номер текущей цепочки, текущего
правила
	);
	MfstState(
		short position, // позиция на ленте
		short pnrule,
		short pnRuleChain // номер текущей цепочки, текущего
правила
	);
	MfstState(short pposition, MFSTSTSTACK pst, short pnrule, short
pnrulechain);
};
struct Mfst // магазинный автомат
{
	enum RC_STEP // код возврата функии step
	{
		NS_OK, // найдено правило и цепочка, цепочка записана в
стек
		NS_NORULE, // не найдено правило грамматики (ошибка
грамматики)
		NS_NORULECHAIN, // не найдена подходящая цепочка правила
(ошибка в исходном коде)
		NS_ERROR, // неизвестный нетерминальный символ грамматики
		TS_OK, // тек. символ ленты == вершине стека, продвинулась
лента, pop стека
		TS_NOK, // тек. символ ленты != вершине стека,
восстановленно состояние
		LENTA_END,// текущая позиция ленты >= lentaSize
		SURPRISE// неожиданный код возврата(ошибка в step)
	};
	struct MfstDiagnosis // диагностика
	{
		short lentaPosition; // позиция в ленте
		RC_STEP rcStep; // код завершения шага
		short nrule; // номер правила
		short nruleChain; // номер цепочки правила
		MfstDiagnosis();
		MfstDiagnosis( // диагностика
			short plentaPosition, // позиция на ленте
```

```
                RC_STEP rcStep, // код завершения шага
                short pnrule, // номер правила
                short pnruleChain // номер цепочки правила
          );
     } diagnosis[MFST_DIAGN_NUMBER]; // последние самые глубокие
сообщения
     GRABALPHABET* lenta; // перекодированая (TS/NS) лента (из LEX)
     short lentaPosition; // текущая позиция на ленте
     short nrule; // номер текущего правила
     short nruleChain; // номер текущей цепочки, текущего правила
     short lentaSize; // размер ленты
     GRB::Greibach greibach; // грамматика Грейбах
     Lexis::LEX lex; // результат работы лексического анализатора
     MFSTSTSTACK st; // стек автомата
     //std::stack<MfstState> storeState; // стек для сохранения
состояний
     my_stack_MfstState storeState;
     Mfst();
     Mfst(
          Lexis::LEX plex, // результат работы лексического
анализатора
          GRB::Greibach pgrebach // грамматика Грейбах
     );
     char* GetCSt(char* buf); // получить содержимое стека
     char* GetCLenta(char* buf, short pos, short n = 25); // лента:
n символов с pos
     char* GetDiagnosis(short n, char* buf); // получить n-ую строку
диагностики или 0x00
     bool SaveState(std::ostream& out); // сохранить состояние
автомата
     bool ResetState(std::ostream& out); // восстановить состояние
автомата
     bool PushChain(
          GRB::Rule::Chain chain // цепочка правила
     );
     RC_STEP step(std::ostream& out); // выполнить шаг автомата
     bool Start(std::ostream& out, std::ostream& outlog); //
запустить автомат
     bool SaveDiagnosis(
          RC_STEP pprcStep // код завершения шага
     );
     void PrintRules(std::ostream& out); // вывести
последовательность правил
     struct Deducation // вывод
     {
          short size; // количество шагов в выводе
          short* nrules; // номера правил грамматики
          short* nruleChains; // номера цепочек правил грамматики
          Deducation() { size = 0; nrules = 0; nruleChains = 0; };
     } deducation;
     bool SaveDeducation(); // сохранить дерево вывода
};
void SyntacticAnalysis(Lexis::LEX lex, Log::LOG log, std::ostream&
```

```
out);
```

Приложение Е

Разбора синтаксическим анализатором исходного кода

```
Трассировка:
Шаг : Правило              Входная лента                    Стек
0   : S->piFUS                                              S$
0   : SAVESTATE:           1
0   :                                                       piFUS$
1   :                      i(ti){ol;oi;b;dti=i(i);dt        iFUS$
2   :                      (ti){ol;oi;b;dti=i(i);dti        FUS$
3   : F->(P)               (ti){ol;oi;b;dti=i(i);dti        FUS$
3   : SAVESTATE:           2
3   :                      (ti){ol;oi;b;dti=i(i);dti        (P)US$
4   :                      ti){ol;oi;b;dti=i(i);dti=        P)US$
5   : P->ti                ti){ol;oi;b;dti=i(i);dti=        P)US$
5   : SAVESTATE:           3
5   :                      ti){ol;oi;b;dti=i(i);dti=        ti)US$
6   :                      i){ol;oi;b;dti=i(i);dti=l        i)US$
7   :                      ){ol;oi;b;dti=i(i);dti=l;        )US$
8   :                      {ol;oi;b;dti=i(i);dti=l;u        US$
9   : U->{N}               {ol;oi;b;dti=i(i);dti=l;u        US$
9   : SAVESTATE:           4
9   :                      {ol;oi;b;dti=i(i);dti=l;u        {N}S$
10  :                      ol;oi;b;dti=i(i);dti=l;u(        N}S$
11  : N->oI;N              ol;oi;b;dti=i(i);dti=l;u(        N}S$
11  : SAVESTATE:           5
11  :                      ol;oi;b;dti=i(i);dti=l;u(        oI;N}S$
12  :                      l;oi;b;dti=i(i);dti=l;u(i        I;N}S$
13  : I->l                 l;oi;b;dti=i(i);dti=l;u(i        I;N}S$
13  : SAVESTATE:           6
13  :                      l;oi;b;dti=i(i);dti=l;u(i        l;N}S$
14  :                      ;oi;b;dti=i(i);dti=l;u(iv        ;N}S$
15  :                      oi;b;dti=i(i);dti=l;u(ivi        N}S$
16  : N->oI;N              oi;b;dti=i(i);dti=l;u(ivi        N}S$
16  : SAVESTATE:           7
16  :                      oi;b;dti=i(i);dti=l;u(ivi        oI;N}S$
17  :                      i;b;dti=i(i);dti=l;u(ivi)        I;N}S$
18  : I->i                 i;b;dti=i(i);dti=l;u(ivi)        I;N}S$
18  : SAVESTATE:           8
18  :                      i;b;dti=i(i);dti=l;u(ivi)        i;N}S$
19  :                      ;b;dti=i(i);dti=l;u(ivi){        ;N}S$
20  :                      b;dti=i(i);dti=l;u(ivi){o        N}S$
21  : N->b;N               b;dti=i(i);dti=l;u(ivi){o        N}S$
…
1351:                      oi;}}}}}}}}}}}}}}}}}}}}}}        N}$
1352: N->oI;               oi;}}}}}}}}}}}}}}}}}}}}}}        N}$
1352: SAVESTATE:           125
1352:                      oi;}}}}}}}}}}}}}}}}}}}}}}        oI;}$
1353:                      i;}}}}}}}}}}}}}}}}}}}}}}         I;}$
1354: I->i                 i;}}}}}}}}}}}}}}}}}}}}}}         I;}$
1354: SAVESTATE:           126
```

```
1354:                        i;}}}}}}}}}}}}}}}}}}}}}}      i;}$
1355:                        ;}}}}}}}}}}}}}}}}}}}}}}       ;}$
1356:                        }}}}}}}}}}}}}}}}}}}}}}        }$
1357:                        }}}}}}}}}}}}}}}}}}}}}}        $
1358: LENTA_END
```

Принятые правила разбора цепочек

```
0    : S->piFUS
2    : F->(P)
3    : P->ti
6    : U->{N}
7    : N->oI;N
8    : I->l
10   : N->oI;N
11   : I->i
13   : N->b;N
15   : N->dti=E;N
19   : E->iK
20   : K->(W)
21   : W->i
24   : N->dti=E;N
28   : E->l
30   : N->u(C){L}
32   : C->ivi
37   : L->oI;N
38   : I->i
40   : N->oI;N
41   : I->l
43   : N->i=E;
45   : E->iM
…
220  : N->w(C){L}!{L}N
222  : C->ivl
227  : L->qI;
228  : I->l
233  : L->oI;
234  : I->l
237  : N->dti=E;N
241  : E->iK
242  : K->(W)
243  : W->l,W
245  : W->l
248  : N->dti=E;N
252  : E->lM
253  : M->vE
254  : E->lM
255  : M->vE
256  : E->l
258  : N->oI;
259  : I->i
```

Приложение Ж

Изменённое представление промежуточного кода

```
0001| pi(ti){ol;
0002| oi;
0003| b;
0004| dti=i@1#;
0005| dti=l;
0006| u(ivi){oi;
0007| ol;
0008| i=ilv;
0009| }}tfi(ti){dti;
0010| dti;
0011| w(ivl){i=l;
0012| }w(ivl){i=l;
0013| }!{i=ilv;
0014| i=ii@1v#;
0015| }ri;
0016| }m{dti;
0017| i=llv;
0018| dti=llv;
0019| dti=ln;
0020| dti=l;
0021| qi;
0022| oi;
0023| b;
0024| qi;
0025| qi;
0026| dti=l;
0027| qi;
0028| ql;
0029| dti=l;
0030| qi;
0031| i=ilv;
0032| qi;
0033| i(i);
0034| ql;
0035| dti=l@1#;
0036| qi;
0037| i=i@1#;
0038| qi;
0039| w(ivl){ql;
0040| }!{ol;
0041| }dti=ll@2##;
0042| }
```

Приложение 3

Графический материал. Граф дерева разбора.

Литература

1) Кодировка ASCII > Таблица символов 1251 (ANSI, WIN) [Электронный ресурс]; Режим доступа: https://tools.otzyvmarketing.ru/blog/poleznoe/Osnova-osnov-kodirovka-ASCII-i-ee-sovremennye-interpretacii-

2) Ахо, А. Теория синтаксического анализа, перевода и компиляции /А. Ахо, Дж. Ульман. – Москва : Мир, 1998. – Т. 2 : Компиляция. - 487 с

3) Парадигмы программирования [Электронный ресурс]; Режим доступа: http://progopedia.ru/paradigm/

4) Герберт, Ш. Справочник программиста по C/C++ / Шилдт Герберт. -3-3 издание – Москва: Вильямс, 2003. -429 с.