МИНИСТЕРСТВО ОБРАЗОВАНИЯ РЕСПУБЛИКИ БЕЛАРУСЬ

Учреждение образования «БЕЛОРУССКИЙ ГОСУДАРСТВЕННЫЙ ТЕХНОЛОГИЧЕСКИЙ УНИВЕРСИТЕТ»

Факультет Информационных Технологий
Кафедра Программной инженерии
Специальность 1-40 01 01 Программное обеспечение информационных технологий
Специализация Программирование интернет-приложений

ПОЯСНИТЕЛЬНАЯ ЗАПИСКА
К КУРСОВОМУ ПРОЕКТУ НА ТЕМУ:

«Разработка компилятора KDA-2022»

Выполнил студент Картузов Данила Александрович
(Ф.И.О. студента)

Руководитель проекта асс. Мущук Артур Николаевич
(учен. степень, звание, должность, подпись, Ф.И.О.)

Заведующий кафедрой к.т.н., доц. Пацей Наталья Владимировна
(учен. степень, звание, должность, подпись, Ф.И.О.)

Консультанты асс. Мущук Артур Николаевич
(учен. степень, звание, должность, подпись, Ф.И.О.)

Нормоконтролер асс. Мущук Артур Николаевич
(учен. степень, звание, должность, подпись, Ф.И.О.)

Курсовой проект защищен с оценкой

Минск 2022

Содержание

Введение

Основной целью данной курсовой работы является разработка транслятора для языка программирования KDA-2022. Язык программирования KDA-2022 предназначен для работы с консолью, выполнения простейших операций над числами. Компилятор KDA-2022 – это программа, задачей которой будет перевод программы, написанной на языке программирования KDA-2022 в программу на язык ассемблера. В данном курсовом проекте трансляция будет осуществляться в код на языке Assembler.

Исходя из цели курсового проекта, были определены следующие задачи:

- разработка спецификации языка программирования;
- разбратка структуры транслятора;
- разработка лексического анализатора;
- разработка синтаксического анализатора;
- разработка семантического анализатора;
- обработка выражений;
- генерация кода на язык Assembler;
- тестирование транслятора.

Решения каждой из указанных задач представлены в соответствующих главах курсового проекта, а именно:

В первой главе определена спецификация языка программирования, т.е. синтаксис и семантика языка.

Во второй главе представлена структура транслятора. Перечислены основные компоненты транслятора, их назначение и принципы взаимодействия, перечень протоколов, формируемых транслятором и содержимое протоколов.

В третьей главе показана работа лексического анализатора, порождающего таблицы лексем и идентификаторов, а также ошибки, которые обрабатывает этот анализатор.

В четвертой главе речь идет о синтаксическом анализаторе, задачей которого является синтаксический разбор текста с распечаткой протокола разбора, дерева разбора на основе таблицы лексем и ошибки, обрабатываемые синтаксический анализатором.

В пятой главе описан семантический анализатор, его работа и обрабатываемые ошибки.

В шестой главе описаны преобразования выражений, допускаемых языком и приведена часть протокола для контрольного примера, которая отображает результаты преобразования выражений в обратную польскую запись.

В седьмой главе описан генератор кода.

1 Спецификация языка программирования

1.1 Характеристика языка программирования

Язык KDA-2022 – это процедурный, строго типизированный, компилируемый, высокоуровневый язык. Не является объектно-ориентированным.

1.2 Определение алфавита языка программирования

Алфавит языка KDA-2022 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}
- специальные символы: {[ ], ( ) , ;, +, -, /, *, >, <, >, =, !, _, ~}
- знаки пунктуации языка: {(), {},[], ;, =}
- пробел, символ табуляции, символ перехода на новую строку

1.3 Применяемые сепараторы

Символы-сепараторы необходимы для разделения операция языка. Сепараторы, используемые в языке программирования KDA-2022, приведены в таблице 1.1.

Таблица 1.1 – Сепараторы

Символ(ы)	Назначение
‘пробел’	Разделитель цепочек. Допускается везде кроме названий идентификаторов и ключевых слов
( )	Параметры операций и функций
{ }	Программный блок инструкций
[ ]	Распознавание индекса массива
,	Разделитель параметров функций
+ - */	Арифметические операции
> < ^ _ ~ !	Логические операции (операции сравнения: больше, меньше, больше или равно, меньше или равно, проверка на равенство, неравенство), используемые в условии цикла/условной конструкции.
;	Разделитель программных конструкций
=	Оператор присваивания

1.4 Применяемые кодировки

Для написания программ язык KDA-2022 использует кодировку Windows-1251, содержащую английский алфавит, а также специальные символы.

	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	Ђ 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

1.5 Типы данных

В языке KDA-2022 есть 2 типа данных: целочисленный беззнаковый и строковый. Описание типов данных, предусмотренных в данном языке представлено в таблице 1.2.

Таблица 1.2 – Типы данных

Тип данных	Описание типа данных
str	Фундаментальный тип данных, используемый для объявления строк. Без явно указанной инициализации переменной присваивается нулевое значение (пустая строка). Используется для работы с символами, каждый из которых занимает 1 байт. Максимальное количество символов – 70.

Продолжение таблицы 1.2

uint	Фундаментальный тип данных, используемый для объявления целочисленных данных. Этот тип данных занимает 4 байта. Без явно указанной инициализации переменной присваивается нулевое значение. Представляет только положительное целое число. Максимальное значение: 2147483647. Минимальное значение: 0. Поддерживает операции: < – меньше > – больше ! – оператор проверки на неравенство ~ – оператор проверки на равенство ^ – больше либо равно _ – меньше либо равно Поддерживает арифметические операции + – сложение - – вычитание / – деление * – умножение

1.6 Преобразование типов данных

В языке программирования KDA-2022 преобразование типов данных не поддерживается. Язык является строго типизированным.

1.7 Идентификаторы

Идентификаторы должны содержать символы латинского алфавита нижнего регистра, цифры. Идентификаторы не должны совпадать с ключевыми словами. Идентификаторы, объявленные внутри функционального блока, получают суффикс, идентичный имени функции, внутри которой они объявлены. Общее количество идентификаторов ограничено максимальным размером таблицы идентификаторов. Максимальная длина идентификатора равна 7 символам.

<буква> ::= a | b | c | d | e | f | g | h | i | j | k | l | m | n | o | p | q | r | s | t | u | v | w | x | y | z

<цифра> ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

<идентификатор> ::= { (<цифра> |<буква>) }

Примеры идентификаторов представлены в таблице 1.3.

Таблица 1.3 – Пример идентификаторов

Идентификатор	Пример
Корректные	str1 num1

Продолжение таблицы 1.3

Некорректные	Var sometextfor print

1.8 Литералы

В языке существует 2 типа литералов: целого, строкового типов. Краткое описание литералов представлено в таблице 1.4.

Таблица 1.4 – Описание литералов

Тип литерала	Описание
Литералы целого типа	Целочисленные литералы, десятичное и восьмеричное представление, инициализируются 0. Литералы только rvalue.
Строковые литералы	Символы, заключённые в ‘ ’, инициализируются пустой строкой, строковые переменные. Максимальное число символов в литерале 70. Только rvalue.

<десятичное число>:: = { 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 }
<восьмеричное число>:: = { 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 }
Примеры идентификаторов представлены в таблице 1.5.

Таблица 1.5 – Пример литералов

Литералы	Пример
Корректные	42 8x16 ‘Hello World’
Некорректные	16x30 83x16 9999999999

1.9 Объявление данных

Для объявления переменной используется ключевое слово var, после которого указывается тип данных и имя идентификатора. Допускается инициализация при объявлении. Для объявления функций используется ключевое слово func, перед которым указывается тип функции. Далее список параметров и тело функции. Все функции должны возвращать значение. Примеры объявление данных представлены в таблице 1.6.

Таблица 1.6 – Пример объявление

Объявление	Пример
объявления целочисленного массива	var [3] uint num1=1,2,3;
объявления переменной целочисленного типа	var uint num1;

Объявления функции	uint func str num(){
	ret 3;
	}

объявления переменной строкового типа	var str str1;

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 и литералы. При объявлении переменные инициализируются значением по умолчанию: для строк это пустая строка, а для беззнакового целого это нуль. Способы инициализации переменных языка программирования KDA-2022 представлены в таблице 1.7.

Таблица 1.7 – Способы инициализации переменных

Вид инициализации	Примечание
var <тип данных> <идентификатор>;	Автоматическая инициализация переменной. uint – инициализируется нулем, str – пустой строкой.
var <тип данных> <идентификатор> = <значение>;	Инициализация переменной с присваиванием значения.
var [литерал] <тип данных> <идентификатор>=<значение>	Инициализация массива

Примеры инициализации данных представлены в таблице 1.8.

Таблица 1.8 – Пример инициализации данных

Инициализация	Пример
целочисленного массива	var [3] uint num1=1,2,3;

Продолжение таблицы 1.8

переменной целочисленного типа	var uint num1=123;
переменной строкового типа	var str str1=’HI’;

1.11 Инструкции языка

Инструкции языка программирования KDA-2022 представлены в общем виде в таблице 1.9.

Таблица 1.9 –Инструкции языка программирования KDA-2022

Инструкция	Запись
Объявление переменной	var <тип данных> <идентификатор>; var <тип данных> <идентификатор> = <значение>; var [литерал] <тип данных> <идентификатор>=<значение>
Присваивание	<идентификатор> = <значение>|<идентификатор>;
Объявление функции	func <тип данных> <идентификатор> ([<тип данных> <идентификатор>][, <тип данных> <идентификатор>]*) {…}

Блок инструкций	{
	…
	}

Возврат из подпрограммы	ret <выражение>;
Вывод данных	print <идентификатор>|<литерал>;
Условный оператор с блоком else	if [<условие>] { … } else { … }
Условный оператор	if [<условие>] { … }

1.12 Операции языка

Язык программирования KDA-2022 может выполнять арифметические операции, представленные в таблице 1.10.

Таблица 1.10 – Операции языка программирования KDA-2022

Операция	Приоритет операции
+	2

Продолжение таблицы 1.10

( ) [ ] > < & ^ _ !	0
,	1
* /	3

1.13 Выражения и их вычисления

Круглые скобки в выражении используются для изменения приоритетов операций. Квадратные скобки предназначены для инициализации размера массива, либо для обращения к элементу массива по индексу.

Не допускается:

- запись двух подряд арифметических операций.
- выполнять арифметических операций с разными типами
- использовать 0 как отрицательный результат логического выражение
- не инициализировать значениями массив
- использовать внутри квадратных скобках идентификатор
- возвращать математические выражение из функции
- использовать идентификатор больше 7 символов

Перед генерацией кода каждое выражение приводится к записи в виде обратной польской записи, для удобства дальнейшего вычисления выражений на языке ассемблера.

1.14 Конструкция языка

Ключевые программные конструкции языка программирования KDA-2022 представлены в таблице 1.11.

Таблица 1.11 – Программные конструкции

Главная функция (точка входа в приложение)	main {…}
Функция	func <тип данных> <идентификатор> ([<тип данных> <идентификатор>][, <тип данных> <идентификатор>]*) { … ret <выражение>;}

Продолжение таблицы 1.11

Условный оператор с блоком else	if [<условие>] { … } else { … }
Условный оператор	if [<условие>] { … }

1.15 Область видимости идентификаторов

В языке программирования KDA-2022 переменные обязаны находиться внутри программного блока функций. Внутри разных областей видимости разрешено объявление переменных с одинаковыми именами. Все переменные, параметры или функции внутри области видимости получают префикс, который отображается в таблице идентификаторов. Объявление глобальных переменных и пользовательских областей не предусмотрено.

1.16 Семантические проверки

Таблица с перечнем семантических проверок, предусмотренных языком, приведена в таблице 1.12.

Таблица 1.12 – Семантические проверки

Номер	Правило
1	Наличие функции main – точка входа в программу
2	Наличие только одной функции входа в программу
3	Идентификаторы не должны повторяться
4	Использование идентификаторов без их объявления
5	Идентификатор должен быть объявлен до его использования.
6	Тип данных переменной должен совпадать с типом значения, которое присваивается этому типу
7	Операнды в выражениях не могут быть разных типов
8	Соответствие типа функции и возвращаемого значения
9	Корректное обращение к элементу массива
10	Корректность математических операций
11	Наличие одной реализации функции

1.17 Распределение оперативной памяти на этапе выполнения

Транслированный код использует две области памяти. В сегмент констант заносятся все литералы. В сегмент данных заносятся переменные и параметры функций. Локальная область видимости в исходном коде определяется за счет использования правил именования идентификаторов и регулируется их префиксами, что и обуславливает их локальность на уровне исходного кода несмотря на то, что в оттранслированным в язык ассемблера коде переменные

имеют глобальную область видимости. В языке программирование KDA-2022 все типы расположены в стеке, что позволяет ускорить работу с данными.

1.18 Стандартная библиотека и её состав

Функции стандартной библиотеки с описанием представлены в таблице 1.13. Стандартная библиотека написана на языке программирования C++.

Таблица 1.13 – Состав стандартной библиотеки

Функция	Описание
char* Date()	Входной параметр: отсутствуют. Выходной параметр: текущее время системы. Строковая функция, возвращает текущее время системы.
char* Time();	Входной параметр: отсутствуют. Выходной параметр: текущее время системы. Строковая функция, возвращает текущее дату системы.

Стандартная библиотека написана на языке C++, подключается к транслированному коду на этапе генерации кода.

Также в стандартной библиотеке реализованы функции для манипулирования выводом. Для вывода предусмотрен оператор print. Эти функции представлены в таблице 1.14.

Таблица 1.14 – Дополнительные функции стандартной библиотеки

Функция на языке C++	Описание
void OutputInt(unsigned int a)	Функция для вывода в стандартный поток значения целочисленного идентификатора/литерала.
void OutputStr(char* ptr)	Функция для вывода в стандартный поток значения строкового идентификатора/литерала.

1.19 Ввод и вывод данных

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

Функции, управляющие вводом/выводом данных, реализованы на языке C++ и вызываются из транслируемого кода, конечному пользователю недоступны. Пользовательская команда print в транслированном коде будут заменена вызовом нужных библиотечных функций. Библиотека, содержащая нужные процедуры, подключается на этапе генерации кода автоматически. Примеры вызова функций вывода представлены в таблице 1.15.

Таблица 1.15 – Вызова функций ввода и вывода

Функции	Пример
OutputInt	print 123;

Продолжение таблицы 1.15

OutputStr	print ‘hello world’;

1.20 Точка входа

В языке KDA-2022 каждая программа должна содержать главную функцию main: точку входа, с которой начнется последовательное выполнение программы.

1.21 Препроцессор

В языке программирования KDA-2022 препроцессор не предусматривается.

1.22 Соглашения о вызовах

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

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

1.23 Объектный код

KDA-2022 транслируется в язык ассемблера.

1.24 Классификация сообщений транслятора

В случае возникновения ошибки в исходном коде программы на языке KDA-2022 и выявлении её транслятором в файл протокола выводится сообщение. Классификация обрабатываемых ошибок приведена в таблице 1.11.

Таблица 1.11 – Классификация сообщений транслятора

Интервал	Описание ошибок
0-99	Системные ошибки
100-199	Ошибки при работе с файлами
200-299	Ошибки лексического анализа
600-999	Ошибки синтаксического анализа
300-599	Ошибки семантического анализа

1.25 Контрольный пример

Контрольный пример представлен в приложении А.

2 Структура транслятора

2.1 Компоненты транслятора, их назначение и принципы взаимодействия

Транслятор преобразует программу, написанную на языке KDA-2022 в программу на языке ассемблера. Компонентами транслятора являются лексический, синтаксический и семантический анализаторы, а также генератор кода на язык ассемблера. Принцип их взаимодействия представлен на рисунке 2.1.

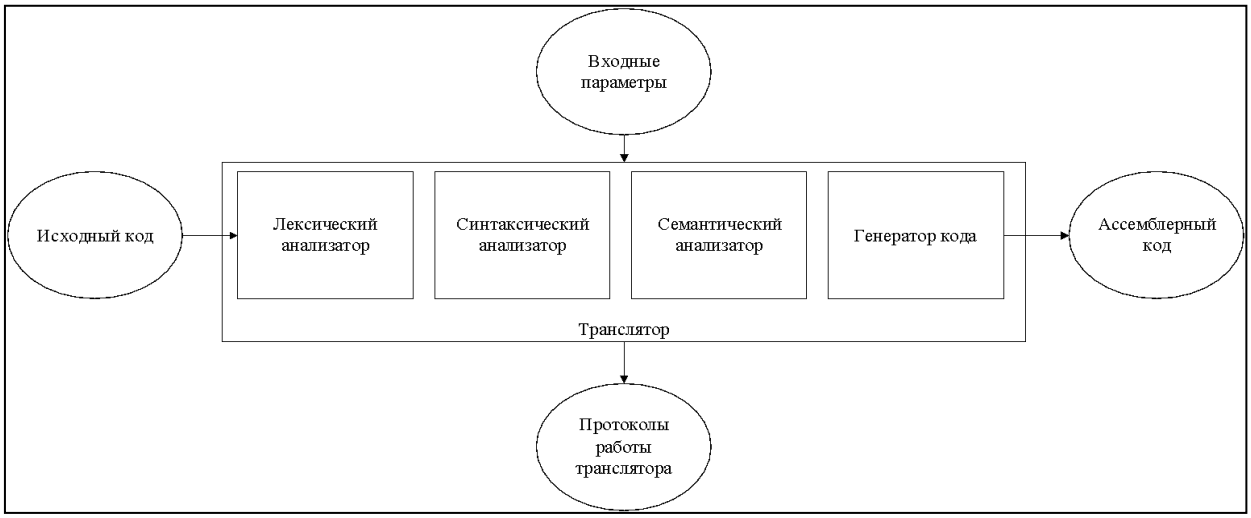

Рисунок 2.1 – Структура транслятора

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

Цели лексического анализатора:

- убрать все лишние пробелы и комментарии;
- 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 транслятора, выполняющая синтаксический анализ, то есть проверку исходного кода на соответствие правилам грамматики. Входной информацией для синтаксического анализа является таблица лексем и таблица идентификаторов. Выходной информацией является дерево разбора

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

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

2.2 Перечень входных параметров транслятора

Входные параметры представлены в таблице 2.1.

Таблица 2.1 – Входные параметры транслятора языка KDA-2022

Входной параметр	Описание	Значение по умолчанию
-in:<имя_файла>	Входной файл с расширением .txt, в котором содержится исходный код на KDA-2022. Это обязательный параметр.	Не предусмотрено
-log:<имя_файла>	Файл с расширение .log, определяет файлы, содержащие результат работы программы:	<имя_файла>.log
-out:<имя_файла>	Файл для записи результата работы транслятора	<имя_файла>.asm

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 KDA-2022

Формируемый протокол	Описание выходного протокола
Файл журнала, заданный параметром "-log:"	Файл с протоколом работы транслятора языка программирования KDA-2022 . 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.
Выходной файл, с расширением "-asm:"	Результат работы программы – файл, содержащий исходный код на языке ассемблера.

Протоколы формируются параллельно с анализаторами.

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 подается исходный код входного языка. Лексический анализатор преобразует исходный текст программы, заменяя лексические единицы языка их внутренним представлением – лексемами.

Для описания лексики языка программирования применяются регулярные грамматики, относящиеся к типу 3 иерархии Хомского. Язык, заданный регулярной грамматикой, называется регулярным языком (типа 3 иерархии Хомского). Регулярный язык однозначно задается регулярным выражением, а распознавателями для регулярных языков являются конечные автоматы.

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 и комментариев. Для облегчения работы синтаксического анализатора

− распознавание идентификаторов и ключевых слов;

− распознавание разделителей и знаков операций.

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

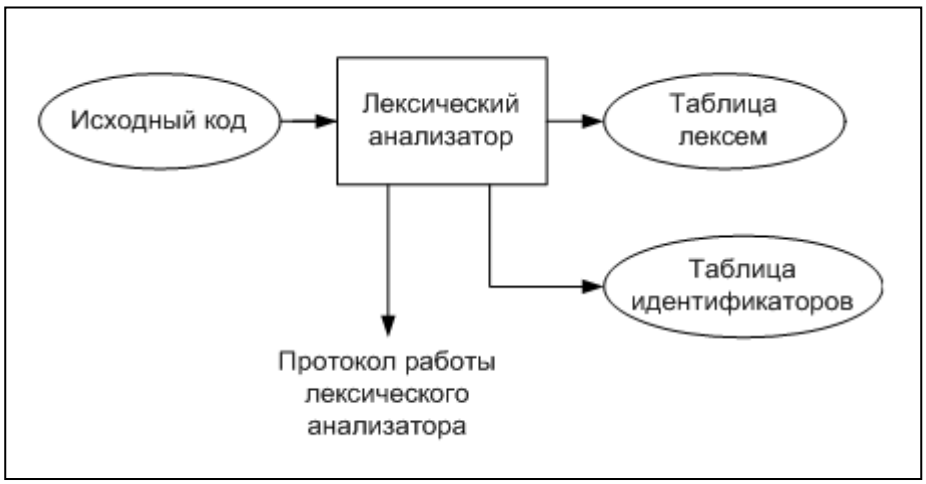

Рисунок 3.1 – Структура лексического анализатора KDA-2022

3.2 Контроль входных символов

Исходный код на языке программирования KDA-2022, прежде чем транслироваться проверяется на допустимость символов.

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::T, IN::ENDL, 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::T, IN::T, IN::L, 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::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, \
}
```

Рисунок 3.2. – Таблица контроля входных символов

Категории входных символов позволяют отличать разрешенные символы от запрещённых и игнорируемых.

Таблица – 3.1 Соответствие символов и их значений в таблице

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

3.3 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:

1. Посимвольно считываем файл с исходным кодом программы;
2. Встреча пробела или знака табуляции является своего рода встречей символа-сепаратора;
3. В отличие от других символов-сепараторов не записываем в очередь лексем пробел и знак табуляции.
4. В цикле проверяем: если следующий символ в потоке будет знаком табуляции или пробела – пропускаем этот символ до тех пор, пока следующим символом будет не \t и не ‘ ‘.

3.4 Перечень ключевых слов

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

Таблица 3.2 – Соответствие токенов и сепараторов с лексемами

Токен	Лексема	Пояснение
uint, str	t	Названия типов данных языка.
Идентификатор	i	Длина идентификатора – 10 символов.
Литерал	l	Литерал любого доступного типа.
func	f	Объявление функции.
ret	r	Выход из функции.
main	m	Главная функция.
var	d	Объявление переменной.
if	?	Разделение конструкций в /условном операторе.
else	e	Разделение конструкций в /условном операторе.
;	;	Разделение выражений.
,	,	Разделение параметров функций.
(	(	Передача параметров в функцию, приоритет операций.
)	)	Закрытие блока для передачи параметров, приоритет операций.
[	[	Передача размера, либо индекса элемента массива
]	]	Закрытие блока для передачи литерала
=	=	Знак присваивания.
+, -, *, /	v	Знаки операций
>, <, ~, !, ^, _	v	Знаки логических операторов

Каждому выражению соответствует 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. Структура конечного автомата изображен на рисунке 3.3.

```
struct RELATION {           // ребро:символ -> вершина графа переходов КА
    char symbol;            // символ перехода
    short nnode;            // номер смежной вершины
    RELATION(
        char c = 0x00,      // символ перехода
        short ns = NULL     // новое состояние
    );
};

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


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

Рисунок 3.3 – Структура конечного автомата

Пример графа перехода конечного автомата изображен на рисунке 3.4.

```
#define UINT(string)        {   string, \
                            5, \
                            FST::NODE(1, FST::RELATION('u', 1)), \
                            FST::NODE(1, FST::RELATION('i', 2)), \
                            FST::NODE(1, FST::RELATION('n', 3)), \
                            FST::NODE(1, FST::RELATION('t', 4)), \
                            FST::NODE() \
}
```

Рисунок 3.4 – Реализации графа конечного автомата для токена uint

3.5 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 (line), номер в таблице идентификаторов, если лексема является идентификатором (idxTI) и приоритет, если лексема является операцией. Таблица идентификаторов содержит имя идентификатора (id), номер в таблице лексем (idxfirstLE), тип данных (iddatatype), тип идентификатора (idtype) и значение (или параметры функций) (value). Код со структурой таблицы лексем представлен на листинге 3.1.

```
struct Entry
{
```

```
      char lexema;
      int sn;
      int idxTI;
};

struct LexTable
{
      int maxsize;
      int size;
      Entry* table;
     };
```

Листинг 3.1 – Структура таблицы лексем

Код со структурой таблицы идентификаторов представлен на листинге 3.2.

```
     enum IDDATATYPE { UINT = 1, STR = 2, UINTARRAY=3};
     enum IDTYPE { V = 1, F = 2, P = 3, L = 4, OP = 5 };

     struct Entry
     {
          int             idxfirstLE;
          char       id[ID_MAXSIZE];
          IDDATATYPE iddatatype;
          IDTYPE          idtype;
          int             numbersystem;
          int             parmsamount;
          int             index;
          char       function[ID_MAXSIZE];
          int             size = 0;
          struct
          {
               unsigned int vuint = 0;
               struct
               {
                    int len;
                    char str[IT_STR_MAXSIZE - 1]{};
               }vstr;
          }value;
     };
     struct IdTable
     {
          int             maxsize;
          int             size;
          Entry* table;   };
```

Листинг 3.2 – Структура таблицы идентификаторов

3.6 Структура и перечень сообщений лексического анализатора

Структура сообщений содержит информацию о номере сообщения, номер строки и позицию, где было вызвано сообщение в исходном коде, информацию об ошибке. Перечень сообщений представлен на листинге 3.3.

```
ERROR_ENTRY(200, "[Лексическая ошибка] Запрещенный символ в исходном файле
(-in)"),
ERROR_ENTRY(201, "[Лексическая ошибка] Размер таблицы лексем превышен"),
                ERROR_ENTRY(202, "[Лексическая ошибка] Переполнение
таблицы лексем"),
                ERROR_ENTRY(203, "[Лексическая ошибка] Размер таблицы
идентификаторов превышен"),
                ERROR_ENTRY(204, "[Лексическая ошибка] Переполнение
таблицы идентификаторов"),
                ERROR_ENTRY(205, "[Лексическая ошибка] Неизвестная
последовательность символов"),
                ERROR_ENTRY(206, "[Лексическая ошибка] Проверти
комментарий"),
                ERROR_ENTRY(207, "[Лексическая ошибка] Запрещённый
литерал, в данном контексте"),
                ERROR_ENTRY(208, "[Лексическая ошибка] Запрещённый index
массива, воспользуйтесь целочисленным литералом"),
                ERROR_ENTRY(209, "[Лексическая ошибка] Использование не
объявленной переменной"),
```

Листинг 3.3 - Сообщения лексического анализатора

3.7 Принцип обработки ошибок

Для обработки ошибок лексический анализатор использует таблицу с сообщениями. При возникновении сообщения лексический анализатор не игнорирует найденную ошибку. 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. Количество ошибок ограничена размером, зарезервировано 100 ошибок для лексического анализатора, используют 9.

3.8 Параметры лексического анализатора

Входным параметром лексического анализатора является исходный текст программы, написанный на языке KDA-2022, а также файл протокола.

3.9 Алгоритм лексического анализа

Алгоритм работы лексического анализа заключается в последовательном распознавании и разборе цепочек исходного кода и заполнение таблиц идентификаторов и лексем. Лексический анализатор производит распознаёт и разбирает цепочки исходного текста программы. Это основывается на работе конечных автоматов, которую можно представить в виде графов. В случае, если

подходящий автомат не был обнаружен, запоминается номер строки, в которой находился этот токен и выводится сообщение об ошибке. Если токен разобран, то дальнейшие действия, которые будут с ним производиться, будут зависеть от того, чем он является. 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.

Описание алгоритма лексического анализа:

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

Распознавание цепочек основывается на работе конечных автоматов. Работу конечного автомата можно проиллюстрировать с помощью графа переходов. Пример графа для цепочки «print» представлен на рисунке 3.5, где S0 – начальное, а S5 – конечное состояние автомата.

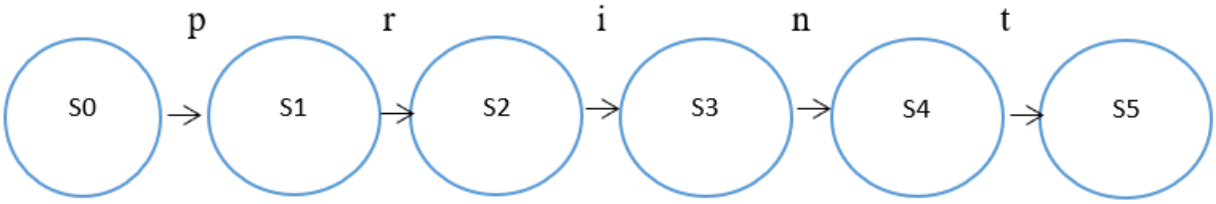

Рисунок 3.5 – Пример графа переходов для цепочки print

Программная реализация разбора цепочки на рисунке 3.6.

```
bool step(FST& fst, short*& rstates)
{
    bool rc = false;
    std::swap(rstates, fst.rstates);
    for (short i = 0; i < fst.nstates; i++)
    {
        if (rstates[i] == fst.position)
            for (int j = 0; j < fst.nodes[i].n_relation; j++)
            {
                if (fst.nodes[i].relations[j].symbol == fst.string[fst.position])
                {
                    fst.rstates[fst.nodes[i].relations[j].nnode] = fst.position + 1;
                    rc = true;
                }
            };
    };
    return rc;
};

bool execute(FST fst)
{
    short* rstates = new short[fst.nstates];
    memset(rstates, 0xff, sizeof(short) * fst.nstates);
    short lstring = strlen(fst.string);

    bool rc = true;
    for (short i = 0; i < lstring && rc; i++)
    {
        fst.position++;
        rc = step(fst, rstates);
    }
    delete[] rstates;

    return (rc ? (fst.rstates[fst.nstates - 1] == lstring) : rc);
};
```

Рисунок 3.6 – Пример алгоритма разбора цепочки

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: часть компилятора, выполняющая синтаксический анализ, то есть исходный код проверяется на соответствие правилам грамматики. Входной информацией для синтаксического анализа является таблица лексем и таблица идентификаторов. Выходной информацией– дерево разбора. Описание структуры синтаксического анализатора языка представлено на рисунке 4.1.

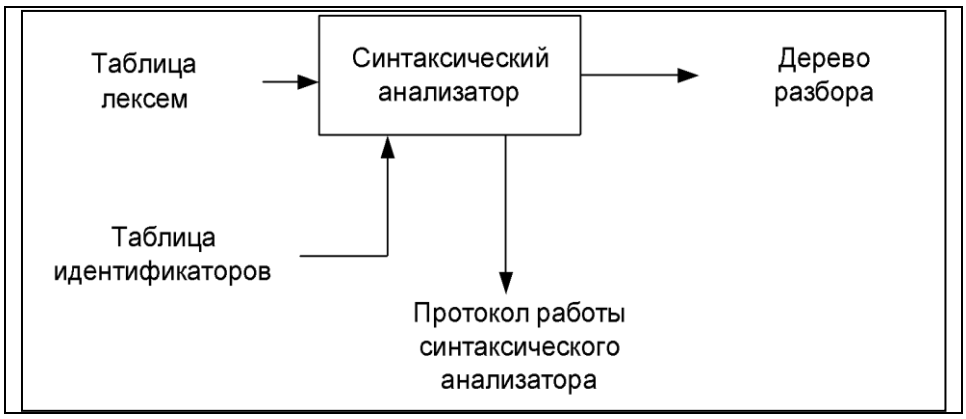

Рисунок 4.1 – Структура синтаксического анализатора

4.2 Контекстно свободная грамматика, описывающая синтаксис языка

В синтаксическом анализаторе транслятора языка KDA-2022 используется контекстно-свободная грамматика типа II в иерархии Хомского (Контекстно-свободная грамматика) $G = \langle T, N, P, S \rangle$, где:

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

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

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

S – начальный символ грамматики, являющийся нетерминалом.

Эта грамматика имеет нормальную форму Грейбах, т.к. она не леворекурсивная (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 KDA-2022 представлены 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.
Описание нетерминальных символов содержится в таблице 4.1.

Таблица 4.1 Таблица правил переходов нетерминальных символов

Нетерминал	Цепочки правил	Описание
S	tfiFBS m{N} tfiFB	Проверка правильности структуры программы
F	(P) ()	Проверка наличия параметров функции
P	ti ti,P	Проверка на правильность параметров функции при её объявлении
B	{NrI;} {rI;} {N}	Проверка наличия тела функции
I	i l	Проверка на недопустимое выражение

N	d[]ti=J;N
	i[l]=E;N
	d[]ti=J;
	i[l]=E;
	dti;N
	iK;N
	iK;
	dti=E;N
	i=E;N
	?(R){X}N
	?(R){X}e{X}N
	rE;N
	iK;N
	dti;
		Проверка на правильность конструкции в теле функции

N	dti=E;
	i=E;
	?(R){X}
	?(R){X}e{X}
	pI;
	rE;
	iK;
		Проверка на правильность конструкции в теле функции

Продолжение таблицы 4.1

R	i l lvl ivi ivl lvi	Проверка на правильность в условном выражении
K	(W) () [l]	Проверка на правильность вызова функции
J	l l,J	Проверка на правильность объявления элементов массива
E	i l (E) iK iM lM (E)M iKM	Проверка на правильность арифметического выражения
W	i l i,W l,W	Проверка на правильность параметров вызываемой функции
M	vE vEM	Проверка на правильность арифметических действий

X	dti;N
	dti=E;N
	i=E;N
	pI;N
	rE;N
	iK;X
	dti;
	dti=E;
	i=E;
	pI;
	rE;
	iK;
		Проверка на правильность конструкции условном выражение

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 В.

Таблица 4.2 – Описание компонентов магазинного автомата

Компоненты	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.
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 конечного автомата и структуру грамматики Грейбах, описывающей правила языка KDA-2022. Данные структуры представлены в приложении Б..

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) Выбор правила, соответствующая правилу грамматики, 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 на листинге 4.1.

```
ERROR_ENTRY(600, "[Синтаксическая ошибка] Неверная структура программы"),
ERROR_ENTRY(601, "[Синтаксическая ошибка] Отсутствует список параметров
функции"),
ERROR_ENTRY(602, "[Синтаксическая ошибка] Ошибка в параметрах функции"),
ERROR_ENTRY(603, "[Синтаксическая ошибка] Отсутствует тело функции"),
ERROR_ENTRY(604, "[Синтаксическая ошибка] Недопустимое выражение"),
ERROR_ENTRY(605, "[Синтаксическая ошибка] Отсутствует тело условия"),
ERROR_ENTRY(606, "[Синтаксическая ошибка] Неверная конструкция в теле
функции"),
ERROR_ENTRY(607, "[Синтаксическая ошибка] Ошибка в условном выражении"),
ERROR_ENTRY(608, "[Синтаксическая ошибка] Ошибка в вызове функции"),
ERROR_ENTRY(609, "[Синтаксическая ошибка] Ошибка в арифметическом
выражении"),
ERROR_ENTRY(610, "[Синтаксическая ошибка] Ошибка в списке параметров при
вызове функции"),
ERROR_ENTRY(611, "[Синтаксическая ошибка] Ошибка в арифметическом
выражении"),
ERROR_ENTRY(612, "[Синтаксическая ошибка] Неверная конструкция в теле
условия"),
```

Листинг 4.1 – Перечень сообщений синтаксического анализатора

4.7 Параметры синтаксического анализатора и режимы его работы

Входным параметром синтаксического анализатора является таблица лексем, полученная на этапе лексического анализа, а также правила контекстно-свободной грамматики в форме Грейбах.

Результаты работы лексического разбора, а именно дерево разбора и протокол работы автомата с магазинной памятью выводятся в журнал работы программы.

4.8 Принцип обработки ошибок

Обработка ошибок происходит следующим образом:

1. Синтаксический анализатор перебирает все правила и цепочки правила грамматики для нахождения подходящего соответствия с конструкцией, представленной в таблице лексем.

2. Если невозможно подобрать подходящую цепочку, то генерируется соответствующая ошибка.
3. Все ошибки записываются в общую структуру ошибок.
4. В случае нахождения ошибки, после всей процедуры трассировки в протокол будет выведено диагностическое сообщение

В структуре грамматики Грейбах цепочки в правилах расположены в порядке приоритета, самые часто используемые располагаются выше, а те, что используются реже – ниже.

4.9 Контрольный пример

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

5 Разработка семантического анализатора

5.1 Структура семантического анализатора

Семантический анализатор принимает на свой вход результаты работ лексического и синтаксического анализаторов, то есть таблицы лексем, идентификаторов и результат работы синтаксического анализатора, то есть дерево разбора, и последовательно ищет необходимые ошибки. 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.

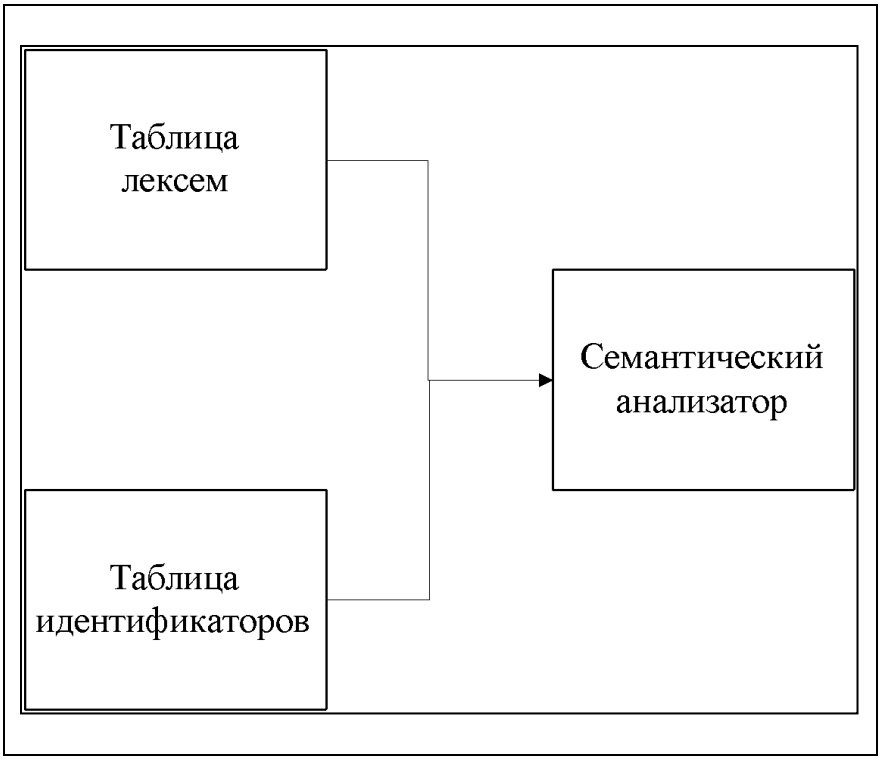

Рисунок 5.1. – Структура семантического анализатора

5.2 Функции семантического анализатора

Семантический анализатор проверяет правильность составления программных конструкций. При невозможности подобрать правило перехода будет выведен код ошибки, а также код этой ошибки. Информация об ошибках выводится в консоль, а также в протокол работы. Функция реализующие проверку семантики языка представлено в таблице 5.1.

Таблица 5.1– Функция реализующие проверку семантики языка

Функция	Описание
Semantic::Parse(lex, log)	Входной параметр: таблица лексем и логер. Выходной параметр: булевое значение. Функция, возвращает информацию о состоянии проверки, семантики языка.

5.3 Структура и перечень сообщений семантического анализатора

Сообщения, формируемые семантическим анализатором, представлены на листинге 5.1.

```
ERROR_ENTRY(300, "[Семантическая ошибка] Имеется не закрытый строковый
литерал"),
ERROR_ENTRY(301, "[Семантическая ошибка] Имеется более одной точки входа в
main"),
ERROR_ENTRY(302, "[Семантическая ошибка] Не имеется точки входа в main"),
ERROR_ENTRY(303, "[Семантическая ошибка] Превышен размер строкового
литерала"),
ERROR_ENTRY(304, "[Семантическая ошибка] Объявление переменной без
ключевого слова var"),
ERROR_ENTRY(305, "[Семантическая ошибка] Необъявленный идентификатор"),
ERROR_ENTRY(306, "[Семантическая ошибка] Объявление  переменной без
указания типа"),
ERROR_ENTRY(307, "[Семантическая ошибка] Попытка реализовать существующую
функцию"),
ERROR_ENTRY(308, "[Семантическая ошибка] Объявление функции без указания
типа"),
ERROR_ENTRY(309, "[Семантическая ошибка] Несовпадение типов передаваемых
параметров функции"),
ERROR_ENTRY(310, "[Семантическая ошибка] Несоответствие арифметических
операторов"),
ERROR_ENTRY(311, "[Семантическая ошибка] Невозможно деление на ноль"),
ERROR_ENTRY(312, "[Семантическая ошибка] Несоответсвие типов данных"),
ERROR_ENTRY(313, "[Семантическая ошибка] Несоответсвие открытых и закрытых
скобок в выражении"),
ERROR_ENTRY(314, "[Семантическая ошибка] Функция возвращает неверный тип
данных"),
ERROR_ENTRY(315, "[Семантическая ошибка] Несоответствие количества
передаваемых параметров функции"),
ERROR_ENTRY(316, "[Семантическая ошибка] Невозможный размер массива"),
ERROR_ENTRY(317, "[Семантическая ошибка] Переполнение масива(увеличьте
размер)"),
ERROR_ENTRY(318, "[Семантическая ошибка] Некорректный индекс массива"),
ERROR_ENTRY(319, "[Семантическая ошибка] Деление на 0"),
ERROR_ENTRY(320, "[Семантическая ошибка] Некорректное обращение к элементу
массива"),
```

Листинг 5.1 – Перечень сообщений семантического анализатора

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.

Функционал транслятора, который реализует определения ошибки, представлен на рисунке 5.2.

```
ERROR geterror(int id)
{
    if (id > 0 && id < ERROR_MAX_ENTRY)
    {
        return errors[id];
    }
    else
    {
        return errors[0];
    }
}
ERROR geterrorin(int id, int line = -1, int col = -1)
{
    if (id > 0 && id < ERROR_MAX_ENTRY)
    {
        errors[id].inext.col = col;
        errors[id].inext.line = line;
        return errors[id];
    }
    else
    {
        return errors[0];
    }
}
```

Рисунок 5.2 – Функция для получения ошибки

5.5 Контрольный пример

Соответствие примеров некоторых ошибок в исходном коде и диагностических сообщений об ошибках приведено в таблице 5.2.

Таблица 5.2 – Примеры диагностики ошибок

Исходный код	Текст сообщения

{
var uint x;
var uint y;
y = 10;
x = y - 50;
}
	Ошибка 302: [Семантическая ошибка] Не имеется точки входа в main

main
{
var uint x;
var str string;
string = x;
}
	Ошибка 312: [Семантическая ошибка] Несоответсвие типов данных
	Строка 5 позиция -1

Продолжение таблицы 5.2

uint func same()
{
  ret 0;
}
 	
uint func same()
{
  ret 1;
}
 	
main
{
  var uint result = same();
}
	Ошибка 307: [Семантическая ошибка] Попытка реализовать существующую функцию
	Строка 6 позиция -1

Кроме приведенных проверок, KDA-2022 имеет большее количество зарезервированных ошибок

6 Вычисление выражений

6.1 Выражения, допускаемые языком

В языке KDA-2022 допускаются выражения, применимые к целочисленным типам данных. В выражениях поддерживаются арифметические операции, такие как +, -, *, /, логические операции, как>, <, !, ^, [], _, ~ и (), а также вызовы функций как операнды арифметических выражений.

Приоритет операций представлен в таблице 6.1.

Таблица 6.1 – Приоритет операций в языке KDA-2022

Приоритет	Операция
0	(> < ~ ! ^ & _
1	,
2	+ -
3	* /
4	)

6.2 Польская запись и принцип ее построения

Все выражения языка KDA-2022 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.

Таблица 6.2 – Пример преобразования выражения в обратную польскую запись

Исходная строка	Результирующая строка	Стек
y+pow(y)		
+pow(y)	y	
pow(y)	y	+
(y)	y	+

Продолжение таблицы 6.2

y)	y	+
)	yy	+
	yy@1+	

Функция реализующее альтернативный способ записи арифметических выражений представлено в таблице 6.3.

Таблица 6.3 – Функция реализующее альтернативный способ записи арифметических выражений

Функция	Описание
Polish::StartPolish(lex);	Входной параметр: таблица лексем. Выходной параметр: отсутствует. Функция, изменяет переданную таблицу лексем, записывая ее альтернативным способом.

6.3 Программная реализация обработки выражений

Программная реализация алгоритма преобразования выражений к польской записи и видоизменённая таблица лексем представлена в приложении Г.

6.4 Контрольный пример

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

7 Генерация кода

7.1 Структура генератора кода

Генерация объектного кода — это перевод компилятором внутреннего представления исходной программы в цепочку символов выходного языка. На вход генератора подаются таблицы лексем и идентификаторов, на основе которых генерируется файл с ассемблерным кодом. Структура генератора кода KDA-2022 представлена на рисунке 7.1.

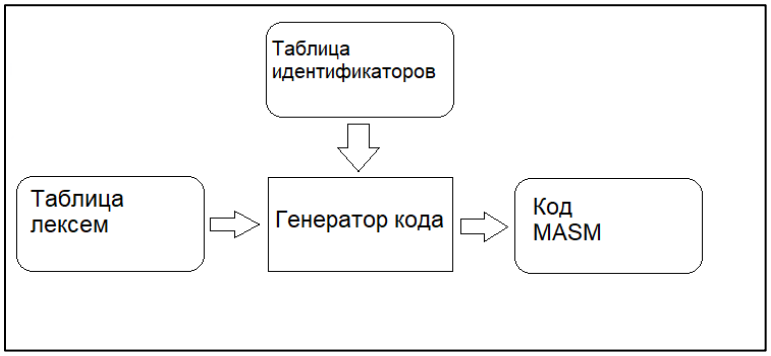

Рисунок 7.1 – Структура генератора кода

Генератор кода последовательно проходит таблицу лексем, при необходимости обращаясь к таблице идентификаторов. В зависимости от пройденных лексем выполняется генерация кода ассемблера.

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 KDA-2022 и на языке ассемблера приведены в таблице 7.1.

Таблица 7.1 – Соответствия типов идентификаторов языка и языка ассемблера

Тип идентификатора на языке KDA-2022	Тип идентификатора на языке ассемблера	Пояснение
uint	dword	Хранит целочисленный тип данных без знака.
str	byte	Каждый символ строки типа str хранится в поле размером 1 байт.

7.3 Статическая библиотека

Статическая библиотека реализована на языке программирования C++. Её реализация находится в проекте StaticLib, в свойствах которого был выбран пункт «статическая библиотека .lib».

В языке программирования KDA-2022 библиотеки подключаются по умолчанию. Подключение библиотеки в языке ассемблера происходит с помощью директивы includelib на этапе генерации кода. Далее с помощью оператора EXTRN объявляются имена 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 указывает ассемблеру тип соответствующего символического имени. Ассемблирование является очень формальной процедурой, то ассемблер должен знать, что представляет из себя каждый символ. Это позволяет ему генерировать правильные команды. Вышеописанное проиллюстрировано на листинге 7.1.

```
out << ".586\n";
out << ".model flat, stdcall\n";

out << "includelib libucrt.lib\n";
out << "includelib kernel32.lib\n";
out << "includelib ../Debug/StaticLib.lib\n";
out << "ExitProcess PROTO :DWORD\n\n";

out << "EXTRN Date: proc\n";
out << "EXTRN Time: proc\n";

out << "EXTRN OutputInt: proc\n";
out << "EXTRN OutputStr: proc\n";

out << "\n.stack 4096\n\n";
```

Листинг 7.1 - Фрагмент функции генерации кода

7.4 Особенности алгоритма генерации кода

В языке KDA-2022 генерация кода строится на основе таблиц лексем и идентификаторов. Общая схема работы генератора кода представлена на рисунке 7.2.

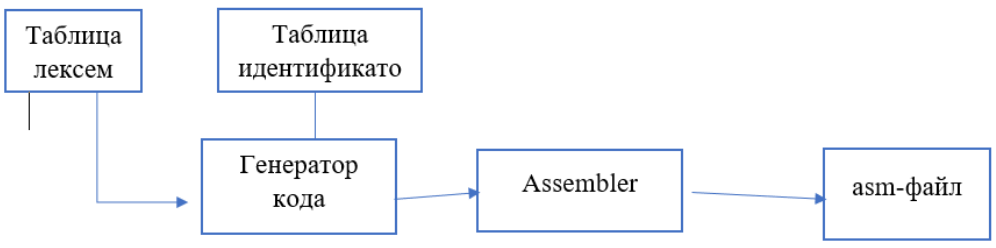

Рисунок 7.2 – Структура генератора кода

7.5 Входные параметры, управляющие генерацией кода

На вход генератору кода поступают таблицы лексем и идентификаторов исходного код программы на языке KDA-2022 . 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 Общие положения

В основе обработки ошибок лежат функции, который принимают в зависимости от перегрузок аргументы и возвращают ошибку с текстом: описывающим ошибку. Функции реализующее обработки ошибок представлено в таблице 8.1.

Таблица 8.1 – Пример функций, обрабатывающие ошибки

Функция	Описание
ERROR geterror(int id)	Входной параметр: id ошибки Выходной параметр: ошибка. Функция ищет по id ошибку в массиве, а затем возвращает её.
ERROR geterrorin(int id, int line=-1, int col=-1)	Входной параметр: id ошибки, строка и колонка, где произошла ошибка. Выходной параметр: ошибка. Функция ищет по id ошибку в массиве, а затем возвращает её, а также ее расположение в исходном тексте.

Информация полученная после обработки ошибка записывается в log.txt. На каждом уровне транслятора может возникнуть ошибка, поэтому для каждого анализатора существуют свои протоколы, куда будет записана информация о успешном разборе, либо об ошибке.

8.2 Результаты тестирования

В языке KDA-2022 обработка ошибок осуществляется на каждом этапе анализа исходного кода.

В языке программирования KDA-2022 не разрешается использовать запрещенные входным алфавитом символы. Результат использования запрещенного символа показан в таблице 8.2.

Таблица 8.2 – Тестирование фазы проверки на допустимость символов

Исходный код	Диагностическое сообщение
uint func ёfelev{ret 11;}	Ошибка 200: [Лексическая ошибка] Запрещенный символ в исходном файле (-in) Строка 1 позиция 9

На этапе лексического анализа могут возникнуть ошибки, описанные в пункте 3.7. Результаты тестирования лексического анализатора показаны в таблице 8.3.

Таблица 8.3 – Тестирование лексического анализатора

Исходный код	Диагностическое сообщение

main
{
  var uint lol;
}
	Ошибка 205: [LEXICAL] Неизвестная последовательность символов
	Строка 3 позиция 12

На этапе синтаксического анализа могут возникнуть ошибки, описанные в пункте 4.6. Результаты тестирования синтаксического анализатора показаны в таблице 8.4.

Таблица 8.4 – Тестирование синтаксического анализатора

Исходный код	Диагностическое сообщение

uint func wr(str s, uint x)
{
  ret x;
}
 	
main
{
  var uint res = wr(9,);
}
	Строка 8, [SYNTAX] Ошибка в списке параметров при вызове функции

Результаты тестирования синтаксического анализатора показаны в таблице 8.5.

Таблица 8.5 – Примеры диагностики ошибок

Исходный код	Текст сообщения

{
  var uint x;
  var uint y;
  y = 10;
  x = y - 50;
}
	Ошибка 302: [Семантическая ошибка] Не имеется точки входа в main

main
{
  var uint x;
  var str string;
  string = x;
}
	Ошибка 312: [Семантическая ошибка] Несоответсвие типов данных
	Строка 5 позиция -1

Продолжение таблицы 8.5

uint func same()
{
  ret 0;
}
 	
uint func same()
{
  ret 1;
}
 	
main
{
  var uint result = same();
}
	Ошибка 307: [Семантическая ошибка]
	Попытка реализовать существующую функцию
	Строка 6 позиция -1

На этапе синтаксического анализа могут возникнуть ошибки, описанные в пункте 5.2.

Заключение

В ходе выполнения курсовой работы был разработан транслятор и генератор кода для языка программирования KDA-2022 со всеми необходимыми компонентами. Таким образом, были выполнены основные задачи данной курсовой работы:

1. Сформулирована спецификация языка KDA-2022;
2. Разработаны конечные автоматы и важные алгоритмы на их основе для эффективной работы лексического анализатора;
3. Осуществлена программная реализация лексического анализатора, распознающего допустимые цепочки спроектированного языка;
4. Разработана контекстно-свободная, приведённая к нормальной форме Грейбах, грамматика для описания синтаксически верных конструкций языка;
5. Осуществлена программная реализация синтаксического анализатора;
6. Разработан семантический анализатор, осуществляющий проверку используемых инструкций на соответствие логическим правилам;
7. Разработан транслятор кода на язык ассемблера;
8. Проведено тестирование всех вышеперечисленных компонентов.

Окончательная версия языка KDA-2022 включает:

1. 2 типа данных;
2. 1 структуру данных;
3. Поддержка операторов ввода и перевода строки;
4. Наличие 4 арифметических операторов для вычисления выражений
5. Наличие 6 логических операторов для использования в условной конструкции
6. Поддержка функций и условий;
7. Наличие библиотеки стандартных функций языка
8. Структурированная и классифицированная система для обработки ошибок пользователя.

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

Список использованных источников

1. Курс лекций по ЯП Наркевич А.С.

2. Ахо, А. Компиляторы: принципы, технологии и инструменты / А. Ахо, Р. Сети, Дж. Ульман. – М.: Вильямс, 2003. – 768с.

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

4. Прата, С. Язык программирования C++. Лекции и упражнения / С. Прата. – М., 2006 — 1104 с.

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

Приложение А

```
uint func cond(uint v,uint k)
{

   if(v>k){
   print 1;
   }else{
   print 2;
   }
    if(v!k){
   print 1;
   }else{
   print 2;
   }

     if(v^k){
   print 1;
   }else{
   print 2;
   }
   ret 1;
}

str func text(){
ret 'text for text';
}

uint func pow(uint c){
var uint res=c*c;
ret res;
}



main
{
   var str date = Date();
   print date;
   var str time = Time();
   print time;
   var uint x;
   var uint y;

   y = 10;
   x = y - 50;
   print x;
   var str pos = text();
   print pos;

   var uint res = cond(4,6);
   print res;
   var uint i = 8x15;
   print i;
   var uint polish;
   var [3]uint arr=4,5,6;
```

```
   polish = y+pow(3)+arr[2];
   print polish;
   arr[1]=23+arr[2];
  polish=arr[1];
  print polish;
}
```

Листинг 1 - Исходный код на языке KDA-2022

```
15.12.2022
12:41:20
4294967256
text for text
2
1
2
1
13
25
29
```

Рисунок 1 – Результат работы

Приложение Б

Таблица лексем

#	Лексема	Строка	Индекс в ТИ
0000	t	1	-
0001	f	1	-
0002	i	1	0
0003	(	1	-
0004	t	1	-
0005	i	1	1
0006	,	1	-
0007	t	1	-
0008	i	1	2
0009	)	1	-
0010	{	2	-
0011	?	4	-
0012	(	4	-
0013	i	4	1
0014	v	4	3
0015	i	4	2
0016	)	4	-
0017	{	4	-
0018	p	5	-
0019	l	5	4
0020	;	5	-
0021	}	6	-
0022	e	6	-
0023	{	6	-
0024	p	7	-
0025	l	7	5
0026	;	7	-
0027	}	8	-
0028	?	9	-
0029	(	9	-
0030	i	9	1
0031	v	9	6
0032	i	9	2
0033	)	9	-
0034	{	9	-
0035	p	10	-
0036	l	10	4
0037	;	10	-
0038	}	11	-
0039	e	11	-
0040	{	11	-
0041	p	12	-
0042	l	12	5

0043	;		12	-
0044	}		13	-
0045	?		15	-
0046	(		15	-
0047	i		15	1
0048	v		15	7
0049	i		15	2
0050	)		15	-
0051	{		15	-
0052	p		16	-
0053	l		16	4
0054	;		16	-
0055	}		17	-
0056	e		17	-
0057	{		17	-
0058	p		18	-
0059	l		18	5
0060	;		18	-
0061	}		19	-
0062	r		20	-
0063	l		20	4
0064	;		20	-
0065	}		21	-
0066	t		23	-
0067	f		23	-
0068	i		23	8
0069	(		23	-
0070	)		23	-
0071	{		23	-
0072	r		24	-
0073	l		24	9
0074	;		24	-
0075	}		25	-
0076	t		27	-
0077	f		27	-
0078	i		27	10
0079	(		27	-
0080	t		27	-
0081	i		27	11
0082	)		27	-
0083	{		27	-
0084	d		28	-
0085	t		28	-
0086	i		28	12
0087	=		28	-
0088	i		28	11
0089	i		28	11
0090	v	*	28	13
0091	;		28	-
0092	r		29	-
0093	i		29	12
0094	;		29	-

0095	}		30	-
0096	m		34	14
0097	{		35	-
0098	d		36	-
0099	t		36	-
0100	i		36	15
0101	=		36	-
0102	@		36	16
0103	0		36	-
0104			36	-
0105	;		36	-
0106	p		37	-
0107	i		37	15
0108	;		37	-
0109	d		38	-
0110	t		38	-
0111	i		38	17
0112	=		38	-
0113	@		38	18
0114	0		38	-
0115			38	-
0116	;		38	-
0117	p		39	-
0118	i		39	17
0119	;		39	-
0120	d		40	-
0121	t		40	-
0122	i		40	19
0123	;		40	-
0124	d		41	-
0125	t		41	-
0126	i		41	20
0127	;		41	-
0128	i		43	20
0129	=		43	-
0130	l		43	21
0131	;		43	-
0132	i		44	19
0133	=		44	-
0134	i		44	20
0135	l		44	23
0136	v	-	44	22
0137	;		44	-
0138	p		45	-
0139	i		45	19
0140	;		45	-
0141	d		46	-
0142	t		46	-
0143	i		46	24
0144	=		46	-
0145	@		46	8
0146	0		46	-

0147			46	-
0148	;		46	-
0149	p		47	-
0150	i		47	24
0151	;		47	-
0152	d		49	-
0153	t		49	-
0154	i		49	25
0155	=		49	-
0156	l		49	26
0157	l		49	27
0158	@		49	0
0159	2		49	-
0160			49	-
0161			49	-
0162	;		49	-
0163	p		50	-
0164	i		50	25
0165	;		50	-
0166	d		51	-
0167	t		51	-
0168	i		51	28
0169	=		51	-
0170	l		51	29
0171	;		51	-
0172	p		52	-
0173	i		52	28
0174	;		52	-
0175	d		53	-
0176	t		53	-
0177	i		53	30
0178	;		53	-
0179	d		54	-
0180	[		54	-
0181	]		54	-
0182	t		54	-
0183	i		54	31
0184	=		54	-
0185	l		54	26
0186	l		54	32
0187	l		54	27
0188			54	-
0189			54	-
0190	;		54	-
0191	i		56	30
0192	=		56	-
0193	i		56	20
0194	l		56	34
0195	@		56	10
0196	1		56	-
0197	v	+	56	33
0198	i		56	31

```
0199 | [                          | 56                          | -
0200 | l                          | 56                          | 5
0201 | ]                          | 56                          | -
0202 | v                     +    | 56                          | 33
0203 |                            | 56                          | -
0204 | ;                          | 56                          | -
0205 | p                          | 57                          | -
0206 | i                          | 57                          | 30
0207 | ;                          | 57                          | -
0208 | i                          | 58                          | 31
0209 | [                          | 58                          | -
0210 | l                          | 58                          | 4
0211 | ]                          | 58                          | -
0212 | =                          | 58                          | -
0213 | l                          | 58                          | 35
0214 | i                          | 58                          | 31
0215 | [                          | 58                          | -
0216 | l                          | 58                          | 5
0217 | ]                          | 58                          | -
0218 | v                     +    | 58                          | 33
0219 | ;                          | 58                          | -
0220 | i                          | 59                          | 30
0221 | =                          | 59                          | -
0222 | i                          | 59                          | 31
0223 | [                          | 59                          | -
0224 | l                          | 59                          | 4
0225 | ]                          | 59                          | -
0226 | ;                          | 59                          | -
0227 | p                          | 60                          | -
0228 | i                          | 60                          | 30
0229 | ;                          | 60                          | -
0230 | }                          | 61                          | -
-------------------------------------------------------------------------
-------------------------------------------------------------------------
--
     Всего лексем: 231
-------------------------------------------------------------------------
--
```

Листинг 1 - Таблица лексем на выходе лексического анализатора

```
-------------------------------------------------------------------------
-----------------
   # | Идентификатор | Тип данных | Тип идентификатора | Индекс в ТЛ
| Значение
-------------------------------------------------------------------------
-----------------
0000 | cond          | uint       | функция            | 2
| –
0001 | condv         | uint       | параметр           | 5
| –
0002 | condk         | uint       | параметр           | 8
| –
```

```
0003 | >              | -          | оператор          | 14
| -
0004 | L1             | uint       | литерал           | 19
| 1
0005 | L2             | uint       | литерал           | 25
| 2
0006 | !              | -          | оператор          | 31
| -
0007 | ^              | -          | оператор          | 48
| -
0008 | text           | str        | функция           | 68
| -
0009 | L3             | str        | литерал           | 73
| [13]"text for text"
0010 | pow            | uint       | функция           | 78
| -
0011 | powc           | uint       | параметр          | 81
| -
0012 | powres         | uint       | переменная        | 86
| 0
0013 | *              | -          | оператор          | 90
| -
0014 | main           | uint       | функция           | 96
| -
0015 | maindate       | str        | переменная        | 100
| [0]""
0016 | Date           | str        | функция           | 102
| -
0017 | maintime       | str        | переменная        | 111
| [0]""
0018 | Time           | str        | функция           | 113
| -
0019 | mainx          | uint       | переменная        | 122
| 0
0020 | mainy          | uint       | переменная        | 126
| 0
0021 | L4             | uint       | литерал           | 130
| 10
0022 | -              | -          | оператор          | 136
| -
0023 | L5             | uint       | литерал           | 135
| 50
0024 | mainpos        | str        | переменная        | 143
| [0]""
0025 | mainres        | uint       | переменная        | 154
| 0
0026 | L6             | uint       | литерал           | 156
| 4
0027 | L7             | uint       | литерал           | 157
| 6
0028 | maini          | uint       | переменная        | 168
| 0
```

```
0029 | L8            | uint       | литерал           | 170
| 13
0030 | mainpolish    | uint       | переменная        | 177
| 0
0031 | mainarr       | unknown    | переменная        | 183
| –
0032 | L9            | uint       | литерал           | 186
| 5
0033 | +             | –          | оператор          | 197
| –
0034 | L10           | uint       | литерал           | 194
| 3
0035 | L11           | uint       | литерал           | 213
| 23
-------------------------------------------------------------------
-----------------
Количество идентификаторов: 36

 ------------------------------------------------------------------
                          -----------------
```

Листинг 2 - Таблица идентификаторов на выходе лексического анализатора

```
#define LEXEMA_FIXSIZE     1
#define LT_MAXSIZE         4096
#define LT_TI_NULLIDX 0xffffffff

#define LEX_UINT      't' // uint
#define LEX_UINTARRAY 't' // uint

#define LEX_STR            't' // str

#define LEX_ID             'i' // identificator
#define LEX_LITERAL        'l' // literal
#define LEX_FUNCTION  'f' // function

#define LEX_MAIN      'm' // main
#define LEX_VAR            'd' // var
#define LEX_RET            'r' // ret
#define LEX_PRINT          'p' // print

#define LEX_IF             '?' // if
#define LEX_ELSE      'e' // else

#define LEX_SEMICOLON ';'   // ;
#define LEX_COMMA          ',' // ,
#define LEX_LEFTBRACE '{' // {
#define LEX_BRACELET  '}' // }
#define LEX_LEFTHESIS '(' // (
#define LEX_RIGHTHESIS     ')' // )
```

```
#define LEX_LEFTSQ          '[' // [
#define LEX_RIGHTSQ         ']' // ]

#define LEX_EQUAL           '=' // =

#define LEX_MORE      'v' // >
#define LEX_LESS      'v' // <
#define    LEX_EQMORE       'v' // ^
#define LEX_EQLESS          'v' // _
#define LEX_EQUALS          'v' // ~
#define LEX_NEQUALS         'v' // !

#define LEX_PLUS      'v' // +
#define LEX_MINUS           'v' // -
#define LEX_STAR      'v' // *
#define LEX_DIRSLASH  'v' // /

#define LEX_OPERATOR 'v'
```

Листинг 3 - Регулярные выражения для лексического распознавателя

Приложение В

```
#include "Main.h"
#include "MFST.h"
#include <stdio.h>
#include <time.h>
namespace MFST {
#pragma region CONSTRUCTORS
MfstState::MfstState()
{
lenta_position = 0;
nrule = -1;
nrulechain = -1;
}

MfstState::MfstState(short pposition, MFSTSTSTACK pst, short pnrulechain)
{
lenta_position = pposition;
st = pst;
nrulechain = pnrulechain;
}

MfstState::MfstState(short pposition, MFSTSTSTACK pst, short pnrule, short
pnrulechain)
{
lenta_position = pposition;
st = pst;
nrule = pnrule;
nrulechain = pnrulechain;
}

Mfst::MfstDiagnosis::MfstDiagnosis()
{
lenta_position = -1;
rc_step = SURPRISE;
nrule = -1;
nrule_chain = -1;
}

Mfst::MfstDiagnosis::MfstDiagnosis(short plenta_position, RC_STEP
prc_step, short pnrule, short pnrule_chain)
{
lenta_position = plenta_position;
rc_step = prc_step;
nrule = pnrule;
nrule_chain = pnrule_chain;
}

Mfst::Mfst() { lenta = 0; lenta_size = lenta_position = 0; }
Mfst::Mfst(LT::LexTable& plex, GRB::Greibach pgreibach)
```

```
{
greibach = pgreibach;
lex = plex;
lenta = new short[lenta_size = lex.size];
for (int k = 0; k < lenta_size; k++)
lenta[k] = GRB::Rule::Chain::T(lex.table[k].lexema);
lenta_position = 0;
st.push(greibach.stbottomT);
st.push(greibach.startN);
nrulechain = -1;
}

#pragma endregion
Mfst::RC_STEP Mfst::step(std::ostream& stream_out)
{

RC_STEP rc = SURPRISE;
if (lenta_position < lenta_size) {
if (GRB::Rule::Chain::isN(st.top())) {
GRB::Rule rule;
if ((nrule = greibach.getRule(st.top(), rule)) >= 0) {
GRB::Rule::Chain chain;
if ((nrulechain = rule.getNextChain(lenta[lenta_position], chain,
nrulechain + 1)) >= 0) {
MFST_TRACE1
savestate(stream_out);
st.pop();
push_chain(chain);
rc = NS_OK;
MFST_TRACE2
}
else {
MFST_TRACE4("NS_NRCHAIN/NS_NR")
saveddiagnosis(NS_NORULECHAIN); rc = resetstate(stream_out) ?
NS_NORULECHAIN : NS_NORULE;
}
}
else rc = NS_ERROR;
}
else if (st.top() == lenta[lenta_position]) {
lenta_position++; st.pop(); nrulechain = -1; rc = TS_OK;
MFST_TRACE3
}
else {
MFST_TRACE4(TS_NOK / NS_NORULECHAIN) rc = resetstate(stream_out) ? TS_NOK
: NS_NORULECHAIN;
}
}
else {
rc = LENTA_END;
MFST_TRACE4(LENTA_END);
}
```

```
return rc;
}

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

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

bool Mfst::resetstate(std::ostream& stream_out)
{
bool rc = false;
MfstState state;
if (rc = (storestate.size() > 0)) {
state = storestate.top();
lenta_position = state.lenta_position;
st = state.st;
nrule = state.nrule;
nrulechain = state.nrulechain;
storestate.pop();
MFST_TRACE5("RESSTATE")
MFST_TRACE2
}

return rc;
}

bool Mfst::saveddiagnosis(RC_STEP prc_step)
{
bool rc = false;
short k = 0;

while (k < MFST_DIAGN_NUMBER && lenta_position <=
diagnosis[k].lenta_position)
k++;

if (rc = (k < MFST_DIAGN_NUMBER)) {
diagnosis[k] = MfstDiagnosis(lenta_position, prc_step, nrule, nrulechain);

for (int i = k + 1; i < MFST_DIAGN_NUMBER; i++)
diagnosis[i].lenta_position = -1;
}
```

```
return rc;
}

bool Mfst::start(std::ostream& stream_out)
{
MFST_TRACE_START
bool rc = false;
RC_STEP rc_step = SURPRISE;
char buf[MFST_DIAGN_MAXSIZE]{};
rc_step = step(stream_out);
double seconds=0;
while (rc_step == NS_OK || rc_step == NS_NORULECHAIN || rc_step == TS_OK
|| rc_step == TS_NOK)
{
clock_t start = clock();
rc_step = step(stream_out);
clock_t end = clock();

seconds += (double)(end - start) / CLOCKS_PER_SEC;
if (seconds > 5) {
exit(-1);
}
}
std::cout << seconds;


switch (rc_step) {
case LENTA_END:
MFST_TRACE4("------>LENTA_END")
stream_out << "-----------------------------------------------------------
------------------------------   ------" << std::endl;
sprintf_s(buf, MFST_DIAGN_MAXSIZE, "%d: всего строк %d, синтаксический
анализ выполнен без ошибок", 0, lex.table[lex.size - 1].sn);
stream_out << std::setw(4) << std::left << 0 << "всего строк " <<
lex.table[lex.size - 1].sn << ", синтаксический анализ выполнен без
ошибок" << std::endl;
rc = true;
break;
case NS_NORULE:
MFST_TRACE4("------>NS_NORULE")
stream_out << "-----------------------------------------------------------
------------------------------   ------" << std::endl;
stream_out << getDiagnosis(0, buf) << std::endl;
stream_out << getDiagnosis(1, buf) << std::endl;
stream_out << getDiagnosis(2, buf) << std::endl;
break;
case NS_NORULECHAIN:
MFST_TRACE4("------>NS_NORULECHAIN") break;
case NS_ERROR:
MFST_TRACE4("------>NS_ERROR") break;
case SURPRISE:
MFST_TRACE4("------>NS_SURPRISE") break;
```

```
}

return rc;
}

char* Mfst::getCSt(char* buf)
{
short p;
for (int k = (signed)st.size() - 1; k >= 0; --k) {
p = st.c[k];
buf[st.size() - 1 - k] = GRB::Rule::Chain::alphabet_to_char(p);
}
buf[st.size()] = '\0';
return buf;
}

char* Mfst::getCLenta(char* buf, short pos, short n)
{
short i = 0, k = (pos + n < lenta_size) ? pos + n : lenta_size;

for (int i = pos; i < k; i++)
buf[i - pos] = GRB::Rule::Chain::alphabet_to_char(lenta[i]);

buf[i - pos] = '\0';
return buf;
}

char* Mfst::getDiagnosis(short n, char* buf)
{
char* rc = new char[200]{};
int errid = 0;
int lpos = -1;
if (n < MFST_DIAGN_NUMBER && (lpos = diagnosis[n].lenta_position) >= 0) {
errid = greibach.getRule(diagnosis[n].nrule).iderror;
Error::ERROR err = Error::geterror(errid);
sprintf_s(buf, MFST_DIAGN_MAXSIZE, "%d: строка %d,    %s", err.id,
lex.table[lpos].sn, err.message);
rc = buf;
}

return rc;
}

void Mfst::printrules(std::ostream& stream_out)
{
MfstState state;
GRB::Rule rule;
for (unsigned short i = 0; i < storestate.size(); i++)
{
state = storestate.c[i];
rule = greibach.getRule(state.nrule);
MFST_TRACE7
```

```
}
}

bool Mfst::savededucation()
{
MfstState state;
GRB::Rule rule;
deducation.nrules = new short[deducation.size = storestate.size()];
deducation.nrulechains = new short[deducation.size];

for (unsigned short i = 0; i < storestate.size(); i++)
{
state = storestate.c[i];
deducation.nrules[i] = state.nrule;
deducation.nrulechains[i] = state.nrulechain;
}

return true;
}

void SyntaxCheck(Lex::LEX lex, Log::LOG log, std::ostream& stream_out)
{
MFST::Mfst mfst(lex.lextable, GRB::getGreibach());
if (!mfst.start(stream_out))
{
std::cout << "Синтаксические ошибки. Для более подробной информации
откройте лог файл.";
exit(-1);
}
mfst.savededucation();
mfst.printrules(stream_out);
}
}
```

Листинг 1 – Структура магазинного автомата

```
#pragma once
#include "GRB.h"
#define GRB_ERROR_SERIES 600
#define NS(n) GRB::Rule::Chain::N(n)
#define TS(n) GRB::Rule::Chain::T(n)
#define ISNS(n) GRB::Rule::Chain::isN(n)

namespace GRB {
Greibach greibach(
NS('S'), TS('$'),
14,
Rule(
NS('S'), GRB_ERROR_SERIES + 0,
3,
Rule::Chain(6, TS('t'), TS('f'), TS('i'), NS('F'), NS('B'), 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(
NS('F'), GRB_ERROR_SERIES + 1,
2,
Rule::Chain(3, TS('('), NS('P'), TS(')')),
Rule::Chain(2, TS('('), TS(')'))
),
Rule(
NS('P'), GRB_ERROR_SERIES + 2,
2,
Rule::Chain(2, TS('t'), TS('i')),
Rule::Chain(4, TS('t'), TS('i'), TS(','), NS('P'))
),
Rule(
NS('B'), GRB_ERROR_SERIES + 3,
3,
Rule::Chain(6, TS('{'), NS('N'), TS('r'), NS('I'), TS(';'), TS('}')),
Rule::Chain(5, TS('{'), TS('r'), NS('I'), TS(';'), TS('}')),
Rule::Chain(3, TS('{'), NS('N'), TS('}'))
),
Rule(
NS('I'), GRB_ERROR_SERIES + 4,
2,
Rule::Chain(1, TS('i')),
Rule::Chain(1, TS('l'))
),
Rule(
NS('U'), GRB_ERROR_SERIES + 5,
1,
Rule::Chain(3, TS('{'), NS('N'), TS('}'))
),

Rule(
NS('N'), GRB_ERROR_SERIES + 6,
22,
Rule::Chain(9, TS('d'), TS('['), TS(']'), TS('t'), TS('i'), TS('='),
NS('J'), TS(';'), NS('N')),
Rule::Chain(8, TS('i'), TS('['), TS('l'), TS(']'),TS('='), NS('E'),
TS(';'), NS('N')),
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('?'), TS('('), NS('R'), TS(')'), TS('{'), NS('X'),
TS('}'), NS('N')),
Rule::Chain(12, TS('?'), TS('('), NS('R'), TS(')'), TS('{'), NS('X'),
TS('}'), TS('e'), TS('{'), NS('X'), TS('}'), NS('N')),
Rule::Chain(4, TS('p'), NS('I'), TS(';'), NS('N')),
Rule::Chain(4, TS('r'), NS('I'), TS(';'), NS('N')),
Rule::Chain(4, TS('r'), NS('I'), TS(';'), NS('N')),
```

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

Rule::Chain(8, TS('d'), TS('['), TS(']'), TS('t'), TS('i'), TS('='),
NS('J'), TS(';')),
Rule::Chain(7, TS('i'), TS('['), TS('l'), TS(']'), TS('='), NS('E'),
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('?'), TS('('), NS('R'), TS(')'), TS('{'), NS('X'),
TS('}')),
Rule::Chain(11, TS('?'), TS('('), NS('R'), TS(')'), TS('{'), NS('X'),
TS('}'), TS('e'), TS('{'), NS('X'), TS('}')),
Rule::Chain(3, TS('p'), NS('I'), TS(';')),
Rule::Chain(3, TS('r'), NS('I'), TS(';')),
Rule::Chain(4, TS('r'), NS('I'), TS(';'), NS('N')),

Rule::Chain(3, TS('i'), NS('K'), TS(';'))
),
Rule(
NS('J'), GRB_ERROR_SERIES + 13,
2,
Rule::Chain(1, TS('l')),
Rule::Chain(3, TS('l'), TS(','), NS('J'))

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

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

Rule::Chain(3, TS('i'), TS('v'), TS('i')),
Rule::Chain(3, TS('i'), TS('v'), TS('l')),
Rule::Chain(3, TS('l'), TS('v'), TS('i'))
),
Rule(
NS('K'), GRB_ERROR_SERIES + 8,
3,
Rule::Chain(3, TS('('), NS('W'), TS(')')),
Rule::Chain(2, TS('('), TS(')')),
Rule::Chain(3, TS('['), TS('l'),TS(']'))

),
Rule(
NS('E'), GRB_ERROR_SERIES + 9,
8,
Rule::Chain(1, TS('i')),
Rule::Chain(1, TS('l')),
```

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

Rule::Chain(2, TS('i'), NS('M')),
Rule::Chain(2, TS('l'), NS('M')),
Rule::Chain(4, TS('('), NS('E'), TS(')'), NS('M')),
Rule::Chain(3, TS('i'), NS('K'), NS('M'))
),
Rule(
NS('W'), GRB_ERROR_SERIES + 10,
4,
Rule::Chain(1, TS('i')),
Rule::Chain(1, TS('l')),
Rule::Chain(3, TS('i'), TS(','), NS('W')),
Rule::Chain(3, TS('l'), TS(','), NS('W'))
),
Rule(
NS('M'), GRB_ERROR_SERIES + 11,
2,
Rule::Chain(2, TS('v'), NS('E')),
Rule::Chain(3, TS('v'), NS('E'), NS('M'))
),
Rule(
NS('X'), GRB_ERROR_SERIES + 12,
12,
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(4, TS('p'), NS('I'), TS(';'), NS('N')),
Rule::Chain(4, TS('r'), NS('E'), TS(';'), NS('N')),
Rule::Chain(4, TS('i'), NS('K'), TS(';'), NS('N')),

Rule::Chain(4, TS('d'), TS('t'), TS('i'), TS(';')),
Rule::Chain(6, TS('d'), TS('t'), TS('i'), TS('='), NS('E'), TS(';')),
Rule::Chain(4, TS('i'), TS('='), NS('E'), TS(';')),
Rule::Chain(3, TS('p'), NS('I'), TS(';')),
Rule::Chain(3, TS('r'), NS('E'), TS(';')),
Rule::Chain(3, TS('i'), NS('K'), TS(';'))
)
);
}
```

Листинг 2 – Правила языка KDA-2022

```
0   : S->tfiFBS
3   : F->(P)
4   : P->ti,P
7   : P->ti
10  : B->{NrI;}
11  : N->?(R){X}e{X}N
13  : R->ivi
```

```
18  : X->pI;
19  : I->l
24  : X->pI;
25  : I->l
28  : N->?(R){X}e{X}N
30  : R->ivi
35  : X->pI;
36  : I->l
41  : X->pI;
42  : I->l
45  : N->?(R){X}e{X}
47  : R->ivi
52  : X->pI;
53  : I->l
58  : X->pI;
59  : I->l
63  : I->l
66  : S->tfiFBS
69  : F->()
71  : B->{rI;}
73  : I->l
76  : S->tfiFBS
79  : F->(P)
80  : P->ti
83  : B->{NrI;}
84  : N->dti=E;
88  : E->iM
89  : M->vE
90  : E->i
93  : I->i
96  : S->m{N}
98  : N->dti=E;N
102 : E->iK
103 : K->()
106 : N->pI;N
107 : I->i
109 : N->dti=E;N
113 : E->iK
114 : K->()
117 : N->pI;N
118 : I->i
120 : N->dti;N
124 : N->dti;N
128 : N->i=E;N
130 : E->l
132 : N->i=E;N
134 : E->iM
135 : M->vE
136 : E->l
138 : N->pI;N
139 : I->i
141 : N->dti=E;N
```

```
145 : E->iK
146 : K->()
149 : N->pI;N
150 : I->i
152 : N->dti=E;N
156 : E->iK
157 : K->(W)
158 : W->l,W
160 : W->l
163 : N->pI;N
164 : I->i
166 : N->dti=E;N
170 : E->l
172 : N->pI;N
173 : I->i
175 : N->dti;N
179 : N->d[]ti=J;N
185 : J->l,J
187 : J->l,J
189 : J->l
191 : N->i=E;N
193 : E->iM
194 : M->vE
195 : E->iKM
196 : K->(W)
197 : W->l
199 : M->vE
200 : E->iK
201 : K->[l]
205 : N->pI;N
206 : I->i
208 : N->i[l]=E;N
213 : E->lM
214 : M->vE
215 : E->iK
216 : K->[l]
220 : N->i=E;N
222 : E->iK
223 : K->[l]
227 : N->pI;
228 : I->i
```

Листинг 3 – Трассировка сходного кода, языка KDA-2022

Приложение Г

```
#include "Polish.h"

namespace Polish {
bool PolishNotation(int i, Lex::LEX& lex)
{
std::stack<LT::Entry> stack;
std::queue<LT::Entry> queue;

LT::Entry placeholder_symbol;
placeholder_symbol.idxTI = -1;
placeholder_symbol.lexema = ' ';
placeholder_symbol.sn = lex.lextable.table[i].sn;

LT::Entry function_symbol;
function_symbol.idxTI = LT_TI_NULLIDX;
function_symbol.lexema = '@';
function_symbol.sn = lex.lextable.table[i].sn;
int idx;

int lexem_counter = 0;
int parm_counter = 0;
int lexem_position = i;
char* buf = new char[i];

bool findFunc = false;

for (i; lex.lextable.table[i].lexema != LEX_SEMICOLON; i++,
lexem_counter++)
{
switch (lex.lextable.table[i].lexema)
{
case LEX_ID:
case LEX_LITERAL:
if (lex.idtable.table[lex.lextable.table[i].idxTI].idtype == IT::F)
{
findFunc = true;
idx = lex.lextable.table[i].idxTI;
}
else
{
if (findFunc)
parm_counter++;
queue.push(lex.lextable.table[i]);
}
continue;

case LEX_LEFTHESIS:
stack.push(lex.lextable.table[i]);
continue;
case LEX_LEFTSQ:
```

```
queue.push(lex.lextable.table[i]);
queue.push(lex.lextable.table[i+1]);
lexem_counter++;
i++;
continue;
case LEX_RIGHTSQ:
queue.push(lex.lextable.table[i]);
continue;
case LEX_RIGHTHESIS:
while (stack.top().lexema != LEX_LEFTHESIS)
{
queue.push(stack.top());
stack.pop();
if (stack.empty())
return false;
}

if (!findFunc)
stack.pop();
else {
function_symbol.idxTI = idx;
idx = LT_TI_NULLIDX;
lex.lextable.table[i] = function_symbol;
queue.push(lex.lextable.table[i]);
_itoa_s(parm_counter, buf, 2, 10);
stack.top().lexema = buf[0];
stack.top().idxTI = LT_TI_NULLIDX;
stack.top().sn = function_symbol.sn;
queue.push(stack.top());
stack.pop();
parm_counter = 0;
findFunc = false;
}
continue;

case LEX_OPERATOR:
while (!stack.empty() && lex.lextable.table[i].priority <=
stack.top().priority)
{
queue.push(stack.top());
stack.pop();
}
stack.push(lex.lextable.table[i]);
continue;
}
}

while (!stack.empty())
{
if (stack.top().lexema == LEX_LEFTHESIS || stack.top().lexema ==
LEX_RIGHTHESIS)
return false;
```

```
queue.push(stack.top());
stack.pop();
}

while (lexem_counter != 0){
if (!queue.empty()){
lex.lextable.table[lexem_position++] = queue.front();
queue.pop();
}
else
lex.lextable.table[lexem_position++] = placeholder_symbol;
lexem_counter--;
}
for (int i = 0; i < lexem_position; i++)
{
if (lex.lextable.table[i].lexema == LEX_OPERATOR ||
lex.lextable.table[i].lexema == LEX_LITERAL)
lex.idtable.table[lex.lextable.table[i].idxTI].idxfirstLE = i;
}

return true;
}

void StartPolish(Lex::LEX& lex)
{
for (int i = 0; i < lex.lextable.size; i++)
{
if (lex.lextable.table[i].lexema == '=')
{
PolishNotation(i + 1, lex);
}
}
}
}
```

Листинг 1 - Реализация польской нотации

```
0001| tfi(ti,ti){?(ivi){pl;
0002| }e{pl;
0003| }?(ivi){pl;
0004| }e{pl;
0005| }?(ivi){pl;
0006| }e{pl;
0007| }rl;
0008| }tfi(){rl;
0009| }tfi(ti){dti=iiv;
0010| ri;
0011| }m{dti=@0 ;
0012| pi;
0013| dti=@0 ;
0014| pi;
```

```
0015| dti;
0016| dti;
0017| i=l;
0018| i=ilv;
0019| pi;
0020| dti=@0 ;
0021| pi;
0022| dti=ll@2  ;
0023| pi;
0024| dti=l;
0025| pi;
0026| dti;
0027| d[]ti=lll  ;
0028| i=il@1vi[l]v ;
0029| pi;
0030| i[l]=li[l]v;
0031| i=i[l];
0032| pi;
0033| }
```

Листинг 2 - Таблица лексем после преобразования к польской нотации

Приложение Д

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

EXTRN Date: proc
EXTRN Time: proc
EXTRN OutputInt: proc
EXTRN OutputStr: proc

.stack 4096

.CONST
      L1 DWORD 1
      L2 DWORD 2
      L3 BYTE "text for text", 0
      L4 DWORD 10
      L5 DWORD 50
      L6 DWORD 4
      L7 DWORD 6
      L8 DWORD 13
      L9 DWORD 5
      L10 DWORD 3
      L11 DWORD 23

.data
      buffer BYTE 256 dup(0)
      powres DWORD 0
      maindate DWORD ?
      maintime DWORD ?
      mainx DWORD 0
      mainy DWORD 0
      mainpos DWORD ?
      mainres DWORD 0
      maini DWORD 0
      mainpolish DWORD 0
      mainarr DWORD 3 dup(0)

.code

cond PROC condv : DWORD, condk : DWORD
      mov eax, condv
      cmp eax, condk
      ja m0
      jb m1
      je m1
m0:
      push L1
```

```
      call OutputInt
      jmp e0
m1:
      push L2
      call OutputInt
e0:
      mov eax, condv
      cmp eax, condk
      jne m2
      je m3
      je m3
m2:
      push L1
      call OutputInt
      jmp e1
m3:
      push L2
      call OutputInt
e1:
      mov eax, condv
      cmp eax, condk
      jae m4
      jb m5
      je m5
m4:
      push L1
      call OutputInt
      jmp e2
m5:
      push L2
      call OutputInt
e2:
      push L1
      jmp local0
local0:
      pop eax
      ret
cond ENDP

text PROC
      push offset L3
      jmp local1
local1:
      pop eax
      ret
text ENDP

pow PROC powc : DWORD
      push powc
      push powc
      pop eax
      pop ebx
```

```
      mul ebx
      push eax
      pop powres
      push powres
      jmp local2
local2:
      pop eax
      ret
pow ENDP

main PROC
      call Date
      push eax
      pop maindate
      push maindate
      call OutputStr
      call Time
      push eax
      pop maintime
      push maintime
      call OutputStr
      push L4
      pop mainy
      push mainy
      push L5
      pop ebx
      pop eax
      sub eax, ebx
      push eax
      pop mainx
      push mainx
      call OutputInt
      call text
      push eax
      pop mainpos
      push mainpos
      call OutputStr
      push L6
      push L7
      pop edx
      pop edx
      push L7
      push L6
      call cond
      push eax
      pop mainres
      push mainres
      call OutputInt
      push L8
      pop maini
      push maini
      call OutputInt
```

```
      push L6
      pop eax
      mov [mainarr+0], eax
      push L9
      pop eax
      mov [mainarr+4], eax
      push L7
      pop eax
      mov [mainarr+8], eax
      push mainy
      push L10
      pop edx
      push L10
      call pow
      push eax
      pop eax
      pop ebx
      add eax, ebx
      push eax
      mov eax, 4
      mul L2
      mov esi, eax
      push [mainarr+esi]
      pop eax
      pop ebx
      add eax, ebx
      push eax
      pop mainpolish
      push mainpolish
      call OutputInt
      push L11
      mov eax, 4
      mul L2
      mov esi, eax
      push [mainarr+esi]
      pop eax
      pop ebx
      add eax, ebx
      push eax
      mov eax, 4
      mul L1
      mov esi, eax
pop [mainarr+esi]
      mov eax, 4
      mul L1
      mov esi, eax
      push [mainarr+esi]
      pop mainpolish
      push mainpolish
      call OutputInt
      push 0call ExitProcessmain ENDPend main
```

Листинг 1 – Генерация в asm исходного языка