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 AVD-2022»

Выполнил студент Авдеева Вера Дмитриевна
(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 Н. В.
(учен. степень, звание, должность, подпись, Ф.И.О.)

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

Курсовой проект защищен с оценкой

Минск 2022

Содержание

Введение

Целью курсовой работы является разработка транслятора для языка программирования AVD-2022. Главной задачей транслятора является, перобразование программы, написанной на языке программирования AVD-2022, в программу, которая будет понятна компьютеру. В данном курсовом проекте трансляция будет осуществляться в код на языке Assembler.

Исходя из цели курсового проекта, были определены следующие задачи:

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

Язык программирования AVD-2022 предназначен для выполнения простейших арифметический действий, операций над строками и числами.

1 Спецификация языка программирования

1.1 Характеристика языка программирования

Язык программирования AVD-2022 является процедурным, строго типизированным, не объектно-ориентированным, компилируемым языком.

Процедурный язык программирования — язык высокого уровня, в котором используется метод разбиения программ на отдельные связанные между собой модули — подпрограммы.

Строго типизированный язык программирования — язык, в котором переменные привязаны к конкретным типам данных. Язык не позволяет смешивать в выражениях различные типы и не выполняет автоматические неявные преобразования.

Объектно-ориентированный язык программирования — язык, построенный на принципах объектно-ориентированного программирования. В основе концепции объектно-ориентированного программирования лежит понятие объекта — некой сущности, которая объединяет в себе поля (данные) и методы (выполняемые объектом действия).

Компилируемый язык программирования — язык программирования, исходный код которого преобразуется компилятором в исходный код на другом языке программирования.

1.2 Определение алфавита языка программирования

В основе алфавита AVD-2022 лежит таблица символов Windows-1251. Исходный код AVD-2022 может содержать символы латинского и русского алфавита, цифры десятичной системы счисления от 0 до 9.

Таблица 1.1

Название подгруппы	Символы подгруппы
Символы латинского алфавита	[a-z] && [A-Z] && [а-я]
Специальные и числовые символы	[‘ - >] && [{ - }] && [[-]] && символ “ “ (пробел) :

1.3 Применяемые сепараторы

Язык AVD-2022 разрешает использовать сепараторы, для написания исходного кода, представленные в таблице 1.2.

Таблица 1.2 Символы-сепараторы языка AVD-2022

Символ(ы)	Назначение
‘пробел’	Разделитель цепочек. Допускается везде кроме названий идентификаторов, ключевых слов и инициализации строки

Продолжение таблицы 1.2

Символ(ы)	Назначение
{ … }	Блок функции или условной конструкции
(…)	Блок фактических или формальных параметров функции, а также приоритет арифметических операций
,	Разделитель параметров функций
+ - */	Арифметические операции
> < == !=	Логические операции (операции сравнения: больше, меньше, проверка на равенство, на неравенство), используемые в условных конструкциях.
;	Разделитель программных конструкций
=	Оператор присваивания

1.4 Применяемые кодировки

Для написания исходного кода на языке программирования AVD-2022 используется кодировка Windows-1251, представленная на рисунке 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	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 AVD-2022 есть 2 типа данных: целочисленный беззнаковый и строковый. Пользовательские типы данных не поддерживаются. Описание типов данных, предусмотренных в данном языке представлено в таблице 1.3

Таблица 1.3 – Типы данных языка AVD-2022

Тип данных	Описание типа данных
Строковый тип данных **string**	Фундаментальный тип данных. Используется для работы с символами, каждый из которых занимает 1 байт. Максимальная допустимая длина строки = 255 символов. Инициализация по умолчанию пустой строкой. Операции над данными строкового типа: присваивание строковому идентификатору значения другого строкового идентификатора, строкового литерала или значения строковой функции, а также использование библиотечных функций. Поддерживаемые операции: = (бинарный) – оператор присваивания.
Беззнаковый целочисленный тип данных **unsigned integer**	Фундаментальный тип данных. Используется для работы с числовыми значениями. Занимает 4 байта. Минимальное допустимое значение: 0. Максимальное допустимое значение: 2 147 483 647 Инициализация по умолчанию: значение 0. Поддерживаемые операции: + (бинарный) – оператор сложения; - (бинарный) – оператор вычитания; * (бинарный) – оператор умножения; / (бинарный) – оператор деления; = (бинарный) – оператор присваивания. В качестве условия условного оператора поддерживаются следующие логические операции: > (бинарный) – оператор «больше»; < (бинарный) – оператор «меньше»; == (бинарный) – оператор проверки на равенство; != (бинарный) – оператор проверки на неравенство.

1.6 Преобразование типов данных

В языке программирования AVD-2022 преобразование типов данных не поддерживается, т.к. язык является строго типизированным.

1.7 Идентификаторы

В языке AVD-2022 идентификаторы должны быть составлены только из символов нижнего регистра английского алфавита. Типы идентификаторов: имя переменной или функции, параметр, имя стандартной функции. Идентификатор составляется из букв латинского алфавита от 1 до 12 символов, без пробелов. Максимальная длина идентификатора равна 12 символам. Идентификатор не может совпадать с ключевыми словами.

<идентификатор> ::= <буква> |<буква><идентификатор>
<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 |

1.8 Литералы

С помощью литералов осуществляется инициализация переменных. В языке AVD-2022 существует 2 типа литералов: литералы целого типа и строковые. Литералы целого типа можно задавать используя 2 системы исчисления : десятичную и шестнадцатеричную. Подробное описание литералов языка AVD-2022 представлены в таблице 1.4.

Таблица 1.4 – Описание литералов

Тип литерала	Пояснение	Пример
Строковый	Набор символов алфавита языка, заключенных в двойные кавычки.	declare string str; str = 'Hello world'; 'Hello world' – строковый литерал.
Целочисленный в шестнадцатеричной системе исчисления	Последовательность с началом “0x” продолжающаяся цифрами 0…9 и буквами a...f	declare unsigned integer a; a = 0x2e;
Целочисленный в десятичной системе исчисления	Последовательность цифр 0…9 без знака	declare unsigned integer a; a = 7;

Ограничения на строковые литералы: не могут иметь пробелы.

1.9 Объявление данных

В языке AVD-2022 объявление данных начинается с ключевого слова declare, указывается тип данных и имя идентификатора.

Пример: declare unsigned integer a, declare string b;

Область видимости: сверху вниз, параметры внутри функции, объявления внутри функции видны только внутри функции, объявления переменных за пределами функций и главной функции предусмотрены.

1.10 Инициализация данных

Инициализация переменной происходит после её объявления. Инициализация переменной в момент объявления запрещена.

Например: declare string word; word = “слово”; declare unsigned integer num; num = 5;

1.11 Инструкции языка

Все возможные инструкции языка программирования AVD-2022 представлены в общем виде в таблице 1.5.

Таблица 1.5 – Инструкции языка программирования AVD-2022

Инструкция	Запись на языке AVD-2022
Объявление переменной	declare <тип данных> <идентификатор>;
Точка входа	main { … }
Объявление внешней функции	function <тип данных> <идентификатор> (<тип данных> <идентификатор>, …) {…}
Инициализация переменной	<идентификатор> = <выражение>; Выражением может быть идентификатор, литерал, или вызов функции соответствующего типа.
Возврат значения из подпрограммы	return <идентификатор> | <литерал>;
Вывод данных	print (<идентификатор> | <литерал>);

Условный оператор	if (<условие> | (<идентификатор> | <литерал>)
	{
	…
	}
	Блок else не предусмотрен.

1.12 Операции языка

В языке AVD-2022 предусмотрены следующие операции с данными. Приоритетность операций определяется с помощью (). Операции представлены в таблице 1.6.

Таблица 1.6 Операции языка AVD-2022

Операция	**Описание**
+	Бинарный, суммирование
-	Бинарный, вычитание
*	Бинарный, умножение
/	Бинарный, деление
<	Бинарный, меньше
>	Бинарный, больше
==	Бинарный, равенство

Продолжение таблицы 1.6

Операция	**Описание**
!=	Бинарный, неравенство
<=	Бинарный, меньше или равно
>=	Бинарный, больше либо равно

1.13 Выражения и их вычисления

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

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

Перед генерацией кода каждое выражение приводится к записи в польской записи для удобства дальнейшего вычисления выражения на языке ассемблера.

1.14 Конструкции языка

В языке программирования AVD-2022 предусмотрена одна главная функция и внешние функции. Программные конструкции представлены в таблице 1.7.

Таблица 1.7 Программные конструкции AVD-2022

Внешняя функция	function идентификатор (<тип данных> <идентификатор>, …)
	{
	…
	return <идентификатор / литерал>;
	}
	Область видимости сверху вниз. Все переменные являются локальными.

Главная функция	main
	{
	…
	}
	Область видимости сверху вниз. Переменные являются локальными.

Условная конструкция	if(<идентификатор/литерал> <знак логической операции> <идентификатор/литерал>)
	{
	…
	}

1.15 Область видимости идентификаторов

Область видимости идентификаторов в языке AVD-2022 – локальная внутри программных блоков функций.

Сверху вниз, параметры внутри функции, объявления внутри функции видны только внутри функции, объявления за пределами функций и главной функции допускаются.

1.16 Семантические проверки

Основные семантические правила языка AVD-2022 проверяемые на этапах работы транслятора, представлены в таблице 1.8. Часть семантических проверок выполняется на этапе лексического анализа.

Таблица 1.8 — Семантические правила

Номер	Правило
1	Должна присутствовать точка входа main и только одна
2	Идентификаторы должны быть объявлены до инициализации и использования
3	Не должно быть объявлений идентификаторов с одинаковыми именами в одном и том же блоке кода
4	Присваивать значение идентификатору можно только соответствующего типа
7	Вызов функции обязует использование скобок после ее названия с передачей параметров соответствующих типов или без них
8	Тип возвращаемого функцией значения должен соответствовать типу функции
9	Деление на ноль запрещено
10	Проводить арифметические операции со строковым типом данных запрещено
11	Превышение размеров строковых и целочисленных литералов
12	Выражение должно быть условным

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.

1.18 Стандартная библиотека и её состав

В AVD-2022 присутствует стандартная библиотека. Возможные функции стандартной библиотеки описаны в таблице 1.9.

Таблица 1.9 Стандартная библиотека

Функция	Описание
sravs(идентификатор или литерал, идентификатор или литерал);	Осуществляет лексикографическое сравнение строк. Применима для идентификаторов типа string и строковых литералов.
stepen (идентификатор или литерал, идентификатор или литерал);	Возведение числа в степень. Применима только для идентификаторов типа unsigned integer, числовых литералов.

Стандартная библиотека написана на языке C++, подключается к транслированному коду на этапе генерации кода. Вызовы стандартных функций доступны там же, где и вызов пользовательских функций. Также в стандартной конечному пользователю. Для вывода предусмотрен оператор **print**. Эти функции представлены в таблице 1.10.

Таблица 1.10 Дополнительные функции стандартной библиотеки

Функция на языке C++	Описание
void printu(unsigned int ui)	Функция для вывода в стандартный поток значения целочисленного идентификатора/литерала.
void prints(char* str)	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 выводом данных, реализованы на языке C++ и вызываются из транслированного 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.

1.20 Точка входа

В языке AVD-2022 каждая программа должна содержать главную функцию (точку входа) **main**. Функция точки входа представлена в таблице 1.10.

Таблица 1.10 — Точка входа

Конструкция	Реализация

Главная функция
(точка входа)
	main
	{
	/ программный блок /
	}

1.21 Препроцессор

Препроцессор, принимающий и выдающий некоторые данные на вход транслятору, в языке AVD-2022 отсутствует.

1.22 Соглашения о вызовах

Соглашение о вызовах – это правила передачи управления от вызывающего к вызываемому коду, определяющие способы передачи параметров и результата вычислений, возврат в точку вызова.

В языке AVD-2022 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

Язык программирования AVD-2022 транслируется в язык ассемблера.

1.24 Классификация сообщений транслятора

Генерируемые транслятором сообщения определяют степень его информативности, то есть сообщения транслятора должны давать максимально полную информацию о допущенной пользователем ошибке при написании программы. Классификация ошибок транслятора приведены в таблице 1.11

Таблица 1.11 Классификация ошибок

Номера ошибок	Характеристика
0 – 99	Системные ошибки
100 – 299	Ошибки лексического анализа
300 – 399	Ошибки семантического анализа
600 – 699	Ошибки синтаксического анализа
400-499, 700-999	Зарезервированные коды ошибок

1.25 Контрольный пример

Контрольный пример представлен в приложении А.

2 Структура транслятора

2.1 Компоненты транслятора, их назначение и принципы взаимодействия

В языке AVD-2022 исходный код транслируется в язык Assembler. Транслятор языка разделён на отдельные части, которые взаимодействуют между собой и выполняют отведённые им функции, которые представлены в пункте 2.1. Для того чтобы получить ассемблерный код, используется выходные данные работы лексического анализатора, а именно таблица лексем и таблица идентификаторов. Для указания выходных файлов используются входные параметры транслятора, которые описаны в таблице 2.1. Структура транслятора языка AVD-2022 приведена на рисунке 1.

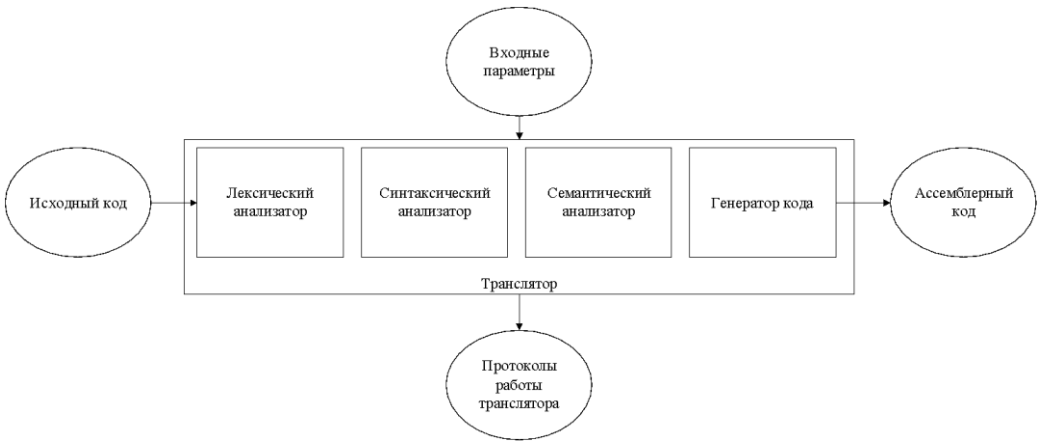

Рисунок 2.1 Структура транслятора языка программирования AVD-2022

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

Цели лексического анализатора:

− убрать все лишние пробелы;

− выполнить распознавание лексем;

− построить таблицу лексем и таблицу идентификаторов;

− при неуспешном распознавании или обнаружении некоторых ошибок во входном тексте выдать сообщение об ошибке.

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

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

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

2.2 Перечень входных параметров транслятора

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

Таблица 2.1 Входные параметры транслятора языка AVD-2022

Входной параметр	Описание параметра	Значение по умолчанию
-in:<путь к in-файлу>	Файл с исходным кодом на языке AVD-2022, имеющий расширение .txt	Не предусмотрено
-log:<путь к log-файлу>	Файл журнала для вывода протоколов работы программы.	Значение по умолчанию: <имя in-файла>.log

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 AVD-2022;

Формируемый протокол	Описание выходного протокола
Файл журнала, заданный параметром "-log:"	Файл с протоколом работы транслятора языка программирования AVD-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.

3 Разработка лексического анализатора

3.1 Структура лексического анализатора

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

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

Функции лексического анализатора:

− удаление «пустых» символов и комментариев. Если «пустые» символы (пробелы, знаки табуляции и перехода на новую строку) и комментарии будут удалены лексическим анализатором, синтаксический анализатор никогда не столкнется с ними (альтернативный способ, состоящий в модификации грамматики для включения «пустых» символов и комментариев в синтаксис, достаточно сложен для реализации);

− распознавание идентификаторов и ключевых слов;

− распознавание констант;

− распознавание разделителей и знаков операций.

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

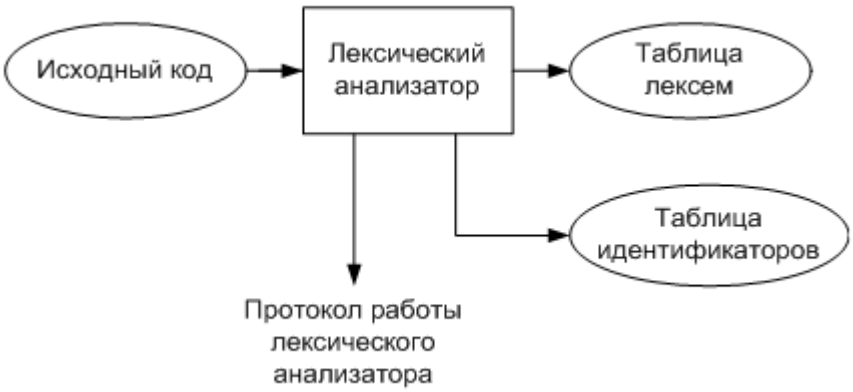

Рисунок 3.1 Структура лексического анализатора

3.2 Контроль входных символов

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

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

Таблица 3.1 Соответствие символов и их значений в таблице

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

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 эти символы, т.е. игнорируем.

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
unsigned integer, string	t	Названия типов данных языка.
Идентификатор	i	Длина идентификатора – 12 символов.
Литерал	l	Литерал любого доступного типа.
function	f	Объявление функции.
return	r	Выход из функции.
main	m	Главная функция.
declare	d	Объявление переменной.
if	u	Условный оператор
;	;	Разделение выражений.
,	,	Разделение параметров функций.
{	{	Начало блока/тела функции.
}	}	Закрытие блока/тела функции.
(	(	Передача параметров в функцию, приоритет операций, условие.
)	)	Закрытие блока для передачи параметров, приоритет операций, условия.
=	=	Знак присваивания.
+ - * /	v	Знаки операций.
> < == !=	s	Знаки логических операторов

Каждому выражению соответствует конечный автомат, по которому происходит разбор данного выражения. Для ключевых слов используется детерминированный конечный автомат, а для идентификаторов и литералов недетерминированный. На каждый автомат в массиве подаётся токен и с помощью регулярного выражения, соответствующего данному графу переходов, происходит разбор. В случае успешного разбора выражения оно записывается в таблицу лексем. Если выражение является идентификатором или литералом, информация также заносится в таблицу идентификаторов. Структура конечного автомата и пример графа перехода конечного автомата находятся в листингах 3.3 и 3.4 соответственно.

```
namespace FST
{
	struct RELATION	// ребро:символ -> вершима 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                // недетерменированный конечный автомат
        {
                char* string;             // цепочка (строка, завершается 0x00)
                short position;           // текущая позиция в цепочке
                short nstates;            // количество состояний автомата
                NODE* nodes;       // граф переходов: [0] - начальное состояние, [nstate-1]
- конечне
                short* rstates;           // возможные состояния автомата на данной позиции
                FST();
                FST(
                        char* s,                  // цепочка (строка, завершается 0x00)
                        short ns,                 // количество состояний автомата
                        NODE n, ...);             // список состояний (граф переходов)
        };
        bool execute(             // выполнить распознавание цепочки
                FST& fst     // недетерминированный конечный автомат
        );
        FST* automat();
}
```

Листинг 3.1 Структура конечного автомата

```
FST* a = new FST(nullptr, 17,
                NODE(2, RELATION('u', 1), RELATION('s', 1)),
                NODE(2, RELATION('n', 2), RELATION('t', 2)),
                NODE(2, RELATION('s', 3), RELATION('r', 3)),
                NODE(2, RELATION('i', 4), RELATION('i', 4)),
                NODE(2, RELATION('g', 5), RELATION('n', 5)),
                NODE(2, RELATION('n', 6), RELATION('g', 16)),
                NODE(1, RELATION('e', 7)),
                NODE(1, RELATION('d', 8)),
                NODE(1, RELATION(' ', 9)),
                NODE(1, RELATION('i', 10)),
                NODE(1, RELATION('n', 11)),
                NODE(1, RELATION('t', 12)),
                NODE(1, RELATION('e', 13)),
                NODE(1, RELATION('g', 14)),
                NODE(1, RELATION('e', 15)),
                NODE(1, RELATION('r', 16)),
                   NODE());
```

Листинг 3.2 Пример реализации графа конечного автомата для токенов unsigned integer и string

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 (sn), и номер в таблице идентификаторов, если лексема является идентификаторов (idxTI). Таблица идентификаторов содержит имя идентификатора (id), номер в таблице лексем (idxfirstLE), тип данных (iddatatype), тип идентификатора (idtype) и значение (или параметры функций) (value). Код C++ со структурой таблицы лексем представлен на листинге 3.3. Код C++ со структурой таблицы идентификаторов представлен на листинге 3.4.

```
struct Entry
	{
		char lexema;
		int sn;
		int idxTI;
		int par = -1;
	};

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

Листинг 3.3 – Код структуры таблицы лексем

```
enum IDDATATYPE { UINT = 1, STR = 2 };
	enum IDTYPE { V = 1, F = 2, P = 3, L = 4, S = 5, U = 6, O = 7 };
	enum SI { Ten = 1, Hex = 2 };

	struct Entry
	{
		int idxfirstLE;
		char id[ID_MAXSIZE];
		IDDATATYPE iddatatype;
		IDTYPE idtype;
		SI si;
		int pars = -1;
		IDDATATYPE* parmstype;
		union
		{
			struct
			{
				unsigned int Ten;
				char Hex[TI_STR_MAXSIZE];
			} vint;
			struct
			{
				char len;
				char str[TI_STR_MAXSIZE];
			} vstr;
		} value;
	};

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

Листинг 3.4 – Код структуры таблицы идентификаторов

3.6 Структура и перечень сообщений лексического анализатора

Для обработки ошибок лексический анализатор использует таблицу с сообщениями. Структура сообщений содержит информацию о номере сообщения, номер строки и позицию, где было вызвано сообщение в исходном коде, информацию об ошибке. При возникновении сообщения, лексический анализатор игнорирует найденную ошибку и продолжает работу с исходным кодом. Перечень сообщений представлен на рисунке 3.5.

```
ERROR_ENTRY(104, "Превышена длина входного параметра"),
ERROR_ENTRY(105, "Превышена длина таблицы идентификаторов"),
ERROR_ENTRY(106, "Превышена длина таблицы лексем"),
ERROR_ENTRY(107, "Превышена длина идентификатора"),
ERROR_ENTRY(108, "Лексема не распознана"),
ERROR_ENTRY(109, "Превышен размер литерала"),
ERROR_ENTRY(315, "Превышен размер строкового литерала"),
ERROR_ENTRY(308, "Превышен размер целочисленного литерала"),
```

Листинг 3.5 – Перечень ошибок лексического анализатора

3.7 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. Если в процессе анализа происходит ошибка, то анализ останавливается.

3.8 Параметры лексического анализатора

Лексический анализатор принимает обработанный и разбитый на отдельные компоненты исходный код на языке AVD-2022. На выходе формируется таблица лексем и таблица идентификаторов.

Результаты работы лексического анализатора, а именно таблицы лексем и идентификаторов выводятся в файл журнала.

3.9 Алгоритм лексического анализа

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

Распознавание цепочек основывается на работе конечных автоматов. Работу конечного автомата можно проиллюстрировать с помощью графа переходов.

Пример графа для цепочки «**string**» представлен на рисунке 3.3, где S0 – начальное, а S6 – конечное состояние автомата.

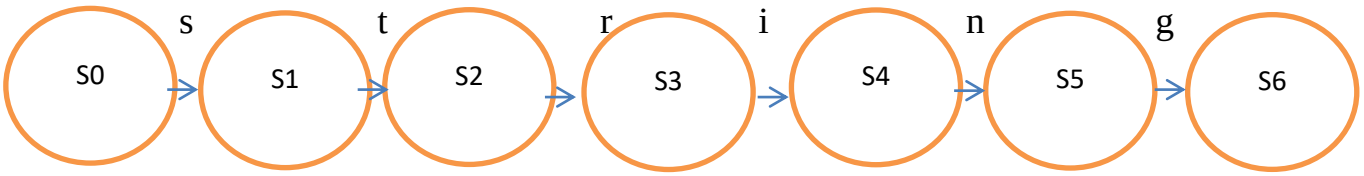

Рисунок 3.3 Пример графа переходов для цепочки string

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.

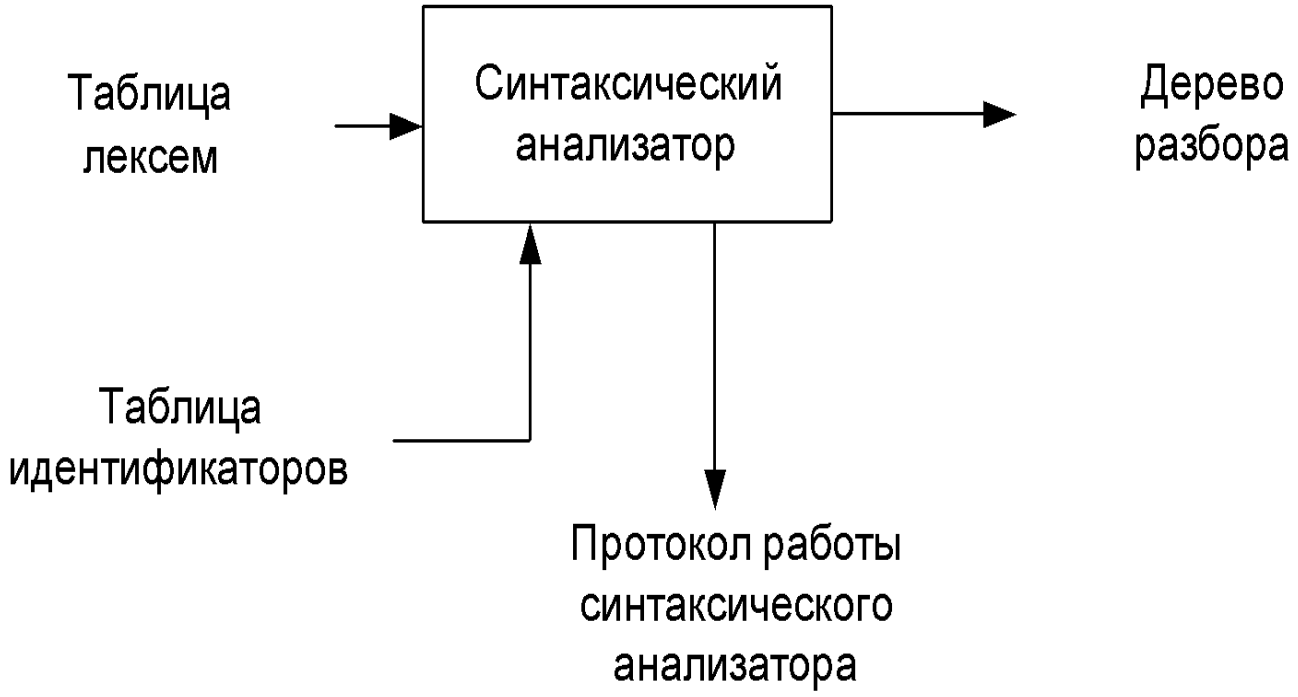

Рисунок 4.1 Структура синтаксического анализатора.

4.2 Контекстно свободная грамматика, описывающая синтаксис языка

В синтаксическом анализаторе транслятора языка AVD-2022 используется контекстно-свободная грамматика $G = \langle T, N, P, S \rangle$, где

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

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

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

S – начальный символ грамматики, являющийся нетерминалом.

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

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

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

Описание нетерминальных символов содержится в таблице 4.1.

Таблица 4.1 Таблица правил переходов нетерминальных символов

Символ	Правила	Какие правила порождает
S	S-> fti(F){NrE}S S->m{N}	Стартовые правила, описывающее общую структуру программы
N	N->dti;N N->dti; N->i=E;N N->i=E; N-> u(E){N}N N->u(E){N} N->dfti(F);N N->dfti(F); N->pi;N N->pi; N->pl;N N->pl; N->pi(E);N N->pi(E);	Правила для операторов
M	M->vE M->vEM M->sE	Правила для подвыражений
F	F->ti,F F->ti	Правила для списка параметров функции
E	E->i E->l E->iM E->(E) E->i(W) E->i(W)M E->lM	Правила для выражений
W	W->i,W W->l,W W->i W->l W->iM,W W->lM,W W->iM W->lM	Правила для параметров вызываемых функций

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

Компонента	Определение	Описание
Q	Множество состояний автомата	Состояние автомата представляет из себя структуру, содержащую позицию на входной ленте, номера текущего правила и цепочки и стек автомата
V	Алфавит входных символов	Алфавит представляет из себя множества терминальных и нетерминальных символов, описание которых содержится в таблица 3.1 и 4.1.
Z	Алфавит специальных магазинных символов	Алфавит магазинных символов содержит стартовый символ и маркер дна стека (представляет из себя символ $)
δ	Функция переходов автомата	Функция представляет из себя множество правил грамматики, описанных в таблице 4.1.
q_0	Начальное состояние автомата	Состояние, которое приобретает автомат в начале своей работы. Представляется в виде стартового правила грамматики
z_0	Начальное состояние магазина автомата	Символ маркера дна стека $
F	Множество конечных состояний	Конечные состояние заставляют автомат прекратить свою работу. Конечным состоянием является пустой магазин автомата и совпадение позиции на входной ленте автомата с размером ленты

4.4 Основные структуры данных

Основные структуры данных синтаксического анализатора включают в себя структуру магазинного автомата, выполняющего разбор исходной ленты, и структуры грамматики Грейбах, описывающей синтаксические правила языка AVD-2022. Данные структуры представлены в приложении В.

4.5 Описание алгоритма синтаксического разбора

Принцип работы автомата следующий:

1) В магазин записывается стартовый символ грамматики;
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 не терминала;

6) Если в магазине встретился не терминал, 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. После 3 исключений синтаксический анализатор завершает свою работу.

4.6 Структура и перечень сообщений синтаксического анализатора

Перечень сообщений синтаксического анализатора представлен на листинге 4.1.

```
ERROR_ENTRY(600,"Неверная структура программы"),
ERROR_ENTRY(601,"Ошибочный оператор"),
ERROR_ENTRY(602,"Ошибка в выражении"),
ERROR_ENTRY(603,"Ошибка в параметрах функции"),
ERROR_ENTRY(604,"Ошибка в параметрах вызываемой функции"),
ERROR_ENTRY(605,"Ошибка в подвыражении"),
ERROR_ENTRY(606, "Неверный синтаксис функции"),
```

Листинг 4.1 – Перечень сообщений синтаксического анализатора

4.7 Параметры синтаксического анализатора и режимы его работы

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

4.8 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 на языке AVD-2022 представлен в приложении В. Дерево разбора исходного кода также представлено в приложении В.

5 Разработка семантического анализатора

5.1 Структура семантического анализатора

Семантический анализатор принимает на свой вход результаты работ лексического и синтаксического анализаторов, то есть таблицы лексем, идентификаторов и результат работы синтаксического анализатора, то есть дерево разбора, и последовательно ищет 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 выполняет проверку на основные правила языка (семантики языка), которые описаны в разделе 1.16. Информация об ошибках выводится в консоль, а также в протокол работы.

5.3 Структура и перечень сообщений семантического анализатора

Сообщения, формируемые семантическим анализатором, представлены на листинге 5.1.

```
ERROR_ENTRY(300, "Необъявленный идентификатор"),
ERROR_ENTRY(301, "Отсутствует точка входа main"),
ERROR_ENTRY(302, "Обнаружено несколько точек входа main"),
ERROR_ENTRY(303, "Попытка переопределения идентификатора"),
ERROR_ENTRY(304, "Превышено максимальное количество параметров функции"),
ERROR_ENTRY(305, "Слишком много параметров в вызове"),
ERROR_ENTRY(306, "Кол-во ожидаемых функцией и передаваемых параметров не
совпадают"),
ERROR_ENTRY(307, "Несовпадение типов передаваемых параметров"),
ERROR_ENTRY(308, "Превышен размер строкового литерала"),
ERROR_ENTRY(309, "Типы данных в выражении не совпадают"),
ERROR_ENTRY(310, "Тип функции и возвращаемого значения не совпадают"),
ERROR_ENTRY(311, "Недопустимое строковое выражение справа от знака ="),
ERROR_ENTRY(312, "Неверное условное выражение"),
ERROR_ENTRY(313, "Деление на ноль"),
```

```
ERROR_ENTRY(314, "Выражение должно быть условным"),
ERROR_ENTRY(315, "Превышен размер целочисленного литерала"),
```

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.

5.5 Контрольный пример

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

Таблица 5.1. Примеры диагностики ошибок

Исходный код	Текст сообщения

main
{
declare number x;
x = 9;
new string y;
y=x;
}
	Ошибка N309: Семантическая ошибка:
	Типы данных в выражении не совпадают
	Строка: 6

main{
new number x; x = 9;
}
main
{ declare string y; y = "qwerty";}
	Ошибка N302: Семантическая ошибка:
	Обнаружено несколько точек входа main
	Строка: 4

6 Преобразование выражений

6.1 Выражения, допускаемые языком

В языке AVD-2022 допускаются вычисления выражений целочисленного типа данных с поддержкой вызова функций внутри выражений. Приоритет операций представлен на таблице 6.1.

Таблица 6.1. Приоритеты операций

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

6.2 Польская запись и принцип ее построения

Выражения в языке AVD-2022 преобразовываются к обратной польской записи.

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

Обратная польская запись – это форма записи математических и логических выражений, в которой операнды расположены перед знаками операций.

Алгоритм построения:

– исходная строка: выражение;

– результирующая строка: польская запись;

– стек: пустой;

– результирующая строка: польская запись;

– исходная строка просматривается слева направо;

– операнды переносятся в результирующую строку в порядке их следования;

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

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

– запятая не помещается в стек, если в стеке операции, то все выбираются в строку;

– отрывающая скобка помещается в стек;

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

– закрывающая скобка с приоритетом, равным 4, выталкивает все до открывающей с таким же приоритетом и генерирует @ (@ – специальный символ, в который записывается информация о вызываемой функции), а в поле приоритета для данной лексемы записывается число параметров вызываемой функции;

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

Использование польской записи позволяет вычислить выражение за один проход.

Таблица 6.2 – Пример преобразования выражения в обратную польскую запись

Исходная строка	Результирующая строка
i=i+i-i+i;	i=iiii+-+;
i=i+l-l;	i=ill+-;
i=i(i,i)+l;	i=ii@2il+;

6.3 Программная реализация обработки выражений

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

6.4 Контрольный пример

В приложении Г приведен результат преобразования выражений в польский формат.

7 Генерация кода

7.1 Структура генератора кода

В языке AVD-2022 генерация кода является заключительным этапом трансляции. Генератор принимает на вход таблицы лексем и идентификаторов, полученные в результате лексического анализа. В соответствии с таблицей лексем строится выходной файл на языке ассемблера, который будет являться результатом работы транслятора. В случае возникновения ошибок генерация кода не будет осуществляться. Структура генератора кода AVD-2022 представлена на рисунке 7.1.

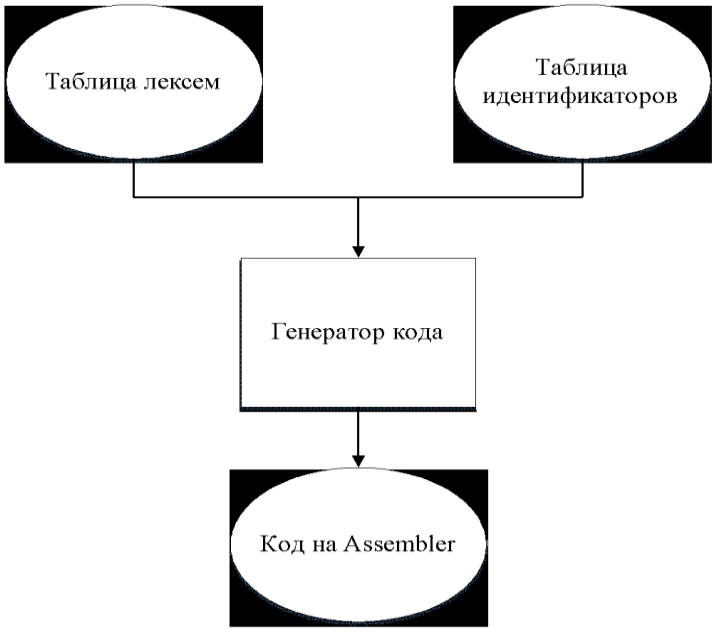

Рисунок 7.1 – Структура генератора кода

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

Таблица 7.1 – Соответствия типов идентификаторов языка AVD-2022 и языка ассемблера

Тип идентификатора на языке AVD-2022	Тип идентификатора на языке ассемблера	Пояснение
unsigned integer	dword	Хранит целочисленный тип данных.
string	dword	Хранит указатель на начало строки. Строка должна завешаться нулевым символом.

7.3 Статическая библиотека

В языке AVD-2022 предусмотрена статическая библиотека. Статическая библиотека содержит функции, написанные на языке C++. Объявление функций статической библиотеки генерируется автоматически в коде ассемблера. Объявление функций статической библиотеки генерируется автоматически.

Таблица 7.2– Функции статической библиотеки

Функция	Назначение
void prints(char* str)	Вывод на консоль строки str
void printu(int ui)	Вывод на консоль целочисленной беззнаковой переменной ui
unsigned int sravs(char* str1, char* str2)	Сравнение строк
unsigned int stepen(unsigned int ui1, unsigned int ui2)	Возведение числа ui1 в степень ui2

7.4 Особенности алгоритма генерации кода

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

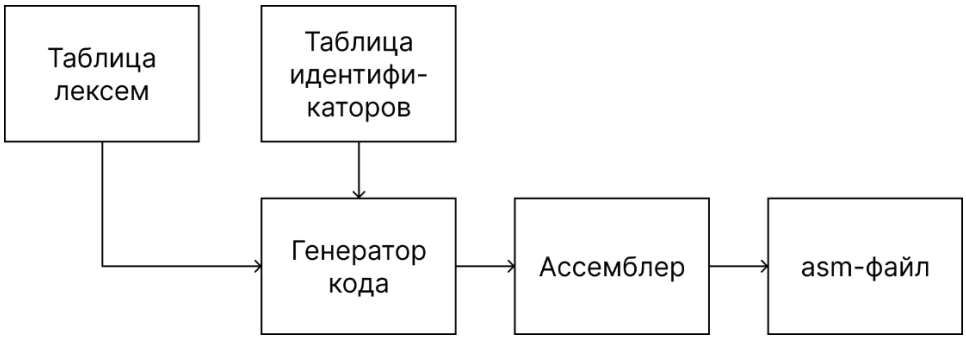

Рисунок 7.2 – Общая схема работы генератора кода

7.5 Входные параметры генератора кода

На вход генератору кода поступают таблицы лексем и идентификаторов исходного код программы на языке AVD-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. Результат работы контрольного примера также приведён в приложении Д.

8 Тестирование транслятора

8.1 Общие положения

В языке AVD-2022 не разрешается использовать запрещённые входным алфавитом символы. Результат использования запрещённого символа показан в таблице 8.1.

Таблица 8.1 – Тестирование фазы проверки на допустимость символов

Исходный код	Диагностическое сообщение

main
{
@
}
	Ошибка 111: Недопустимый символ в исходном файле (-in), строка 3 ,позиция 2.

8.2 Результаты тестирования

На этапе лексического анализа могут возникнуть ошибки, описанные в пункте 3.7.

Результаты тестирования лексического анализатора показаны в таблице 8.2.

Таблица 8.2 – Тестирование лексического анализатора

Исходный код	Диагностическое сообщение
main {declare string x11;}	Ошибка 108: Лексема не распознана, строка: 2

main
{
a = 7;
}
	Ошибка 300: Необъявленный идентификатор, строка 3.

declare unsigned integer s; s = 5 + 4;	Ошибка 301: Отсутствует точка входа main.

main
{
}
main
{
}
	Ошибка 302: Обнаружено несколько точек main.

function unsigned integer lililililililililili(unsigned integer a)
{
…
}
	Ошибка 107: Превышен размер идентификатора

На этапе лексического анализа обрабатываются ошибки, которые препятствую правильному построению таблицы лексем и идентификаторов.

На этапе синтаксического анализа в языке AVD-2022 могут возникнуть ошибки, описанные в пункте 4.6. Результаты тестирования синтаксического анализатора показаны в таблице 8.3.

Таблица 8.3 - Тестирование синтаксического анализатора

Исходный код	Диагностическое сообщение
main new number x; }	Ошибка 600: строка 1, Синтаксическая ошибка: Неверная структура программы
main {string declare id;}	Ошибка 601: строка 2, Ошибочный оператор
function string fi(string id) {declare string ig; ig= +id}	Ошибка 602: строка 2, Ошибка в выражении
function string x(id string){}	Ошибка 603: строка 1, Ошибка в параметрах функции
function string fi(string id, string ig) {return id;} main {declare string str; str=fi(i;i);}	Ошибка 604: строка 3, Ошибка в параметрах вызываемой функции
main { declare unsigned integer num; num=1; declare unsigned integer nums; nums=num++num;}	Ошибка 605: строка 2, Ошибка в подвыражении

Семантический анализ в языке AVD-2022 содержит множество проверок по семантическим правилам, описанным в пункте 1.16. Итоги тестирования семантического анализатора на корректное обнаружение семантических ошибок приведены в таблице 8.4.

Таблица 8.4 -Тестирование семантического анализатора

Исходный код	Диагностическое сообщение
Main{declare unsigned int t; declare string t;}	Ошибка 303: Попытка переопределения идентификатора Строка: 2
function string fi(string x, string y, string z, string s) {} main{return 0;}	Ошибка 304: Превышено максимальное количество параметров функции Строка: 1
function string fi(string str){return 5;} main{return 0;}	Ошибка 310: Тип функции и возвращаемого значения не совпадают Строка: 1

Продолжение таблицы 8.4

Исходный код	Диагностическое сообщение
function string fi(string par){return par;} main{declare string str; str=fi(“a”,”b”,”c”,”d”);}	Ошибка 305: Слишком много параметров в вызове Строка: 1
function string fi(string par){return par;} main{declare string str; str=fi(“a”,”b”,”c”,”d”);}	Ошибка 305: Слишком много параметров в вызове Строка: 1
function string fi(string x){return "a";} main{declare string str; str=fi("a", "b");}	Ошибка 306: Кол-во ожидаемых функцией и передаваемых параметров не совпадают Строка: 2
function string fi(string x){return "a";} main{ declare string str; str=fi("a", "b");}	Ошибка 307: Несовпадение типов передаваемых параметров Строка: 2
Main{declare unsigned integer x; x = 5 + "abc";]	Ошибка 309: Типы данных в выражении не совпадают Строка: 1
main{declare string x; x = "abc" + "d";}	Ошибка 311: Недопустимое строковое выражение справа от знака '=' Строка: 1
main {declare unsigned integer x; x=5; if(lex==”gf”){print 5;}}	Ошибка 312: Семантическая ошибка: Неверное условное выражение Строка: 3
main {declare unsigned integer x; x=5/0}	Ошибка 313: Деление на ноль. Строка: 3
main {declare unsigned integer x; x=5; if(lex+5){print 5;}}	Ошибка 313: Выражение должно быть условным. Строка: 3

Заключение

В ходе выполнения курсовой работы был разработан транслятор для языка программирования AVD-2022. Таким образом, были выполнены основные задачи данной курсовой работы:

- Сформулирована спецификация языка AVD-2022;
- Разработаны конечные автоматы и алгоритмы для реализация лексического анализатора;
- Разработана контекстно-свободная, приведённая к нормальной форме Грейбах, грамматика для описания синтаксически верных конструкций языка;
- Разработан семантический анализатор, осуществляющий проверку смысла используемых инструкций;
- Разработан транслятор с языка программирования AVD-2022 на язык низкого уровня Assembler;
- Проведено тестирование всех вышеперечисленных компонентов.

Окончательная версия языка AVD-2022 включает:

1) 2 типа данных;
2) Поддержка операции вывода;
3) 2 библиотечные функции
4) Возможность вызова функций стандартной библиотеки;
5) Наличие 4 арифметических операторов для вычисления выражений;
6) Наличие 6 операторов сравнения для целочисленных переменных
7) Структурированная система для обработки ошибок пользователя.
8) Условный оператор;

Список использованных источников

1. Курс лекций по предмету «Конструирование программного обеспечения» Наркевич А.С.

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

3. Молчанов, А. Ю. Системное программное обеспечение / А. Ю. Молчанов. – СПб.: Питер, 2010. – 400 с.

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

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

6. Орлов, С.А. Теория и практика языков программирования / С.А. Орлов – 2014. – 689 с.

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

Приложение А

```
function unsigned integer su(unsigned integer a, unsigned integer b,
unsigned integer d)
{
 declare unsigned integer res;
 res = a+b+d;
 return res;
}
main
{
 declare unsigned integer a;
 declare unsigned integer b;
 declare unsigned integer c;
 a = 4;
 b = 0x2a;
 declare function unsigned integer stepen(unsigned integer ui, unsigned
integer us);
 c = stepen(a,2);
 print c;
 declare unsigned integer sum;
 sum = su(a,b,c);
 print sum;
 declare unsigned integer sumo;
 sumo = su(1,a,3);
 print sumo;
 declare string str;
 declare string stro;
 str="exampleExample";
 stro="example";
 declare function unsigned integer sravs(string strm, string strl);
 declare unsigned integer rezs;
 rezs=sravs(str,stro);
if(rezs==1)
{
 print "Okey";
}
if(b!=a)
{
 print "Yes";
}
if(a>=c)
{
 print "Yes";
}
if(b<=c)
{
 print "Yes";
}
}
```

Листинг 1 - Исходный код на языке AVD-2022

Приложение Б

Протокол по работе ТИ - Полная ТИ ->

№	инд первой стр в ТЛ	идентификатор	тип данных	тип идентификатора	значение
0	2	su	1	2	0
1	5	a	1	3	0
2	8	b	1	3	0
3	11	d	1	3	0
4	16	sures	1	1	0
5	21	+	1	7	0
6	34	maina	1	1	0
7	38	mainb	1	1	0
8	42	mainc	1	1	0
9	46	LEX1	1	4	4
10	50	LEX2	1	4	2a
11	55	stepen	1	5	0
12	58	ui	1	3	0
13	61	us	1	3	0
14	70	LEX3	1	4	2
15	78	mainsum	1	1	0
16	96	mainsumo	1	1	0
17	102	LEX4	1	4	1
18	106	LEX5	1	4	3
19	114	mainstr	2	1	
20	118	mainstro	2	1	
21	122	LEX6	2	4	exampleExample
22	126	LEX7	2	4	example
23	131	sravs	1	5	0
24	134	strm	2	3	
25	137	str1	2	3	
26	142	mainrezs	1	1	0
27	156	==	1	6	0
28	157	LEX8	1	4	1
29	161	LEX9	2	4	Okey
30	167	!=	2	6	
31	172	LEX10	2	4	Yes
32	178	>=	2	6	
33	183	LEX11	2	4	Yes
34	189	<=	2	6	
35	194	LEX12	2	4	Yes

Рисунок 1 - Таблица идентификаторов на выходе лексического анализатора

Протокол по работе ТЛ - Полная ТЛ ->

№	номер в табл идентиф	лексема	номер в исходном коде
0	-1	f	1
1	-1	t	1
2	0	i	1
3	-1	(	1
4	-1	t	1
5	1	i	1
6	-1	,	1
7	-1	t	1
8	2	i	1
9	-1	,	1
10	-1	t	1
11	3	i	1
12	-1	)	1
13	-1	{	2
14	-1	d	3
15	-1	t	3
16	4	i	3
17	-1	;	3
18	4	i	4
19	-1	=	4
20	1	i	4
21	5	v	4
22	2	i	4
23	5	v	4
24	3	i	4
25	-1	;	4
26	-1	r	5
27	4	i	5
28	-1	;	5
29	-1	}	6
30	-1	m	7
31	-1	{	8
32	-1	d	9
33	-1	t	9
34	6	i	9
35	-1	;	9
36	-1	d	10
37	-1	t	10
38	7	i	10
39	-1	;	10
40	-1	d	11
41	-1	t	11
42	8	i	11

Рисунок 2 – Часть таблицы лексем

Приложение В

```
Greibach greibach(NS('S'), TS('$'), // стартовый символ //дно стека
            6,
            Rule(NS('S'), GRB_ERROR_SERIES + 0, // неверная структура программы
                  3,
                  Rule::Chain(4, TS('m'), TS('{'), NS('N'), TS('}')),
                  Rule::Chain(13, TS('f'), TS('t'), TS('i'), TS('('), NS('F'), TS(')'), TS('{'),
NS('N'), TS('r'), NS('E'), TS(';'), TS('}'), NS('S')),
                  Rule::Chain(5, TS('m'), TS('{'), NS('N'), TS('}'), NS('S'))
            ),
            Rule(NS('N'), GRB_ERROR_SERIES + 1,  // конструкции в функциях
                  14,
                  Rule::Chain(5, TS('d'), TS('t'), TS('i'), TS(';'), NS('N')),
                  Rule::Chain(4, TS('d'), TS('t'), TS('i'), TS(';')),
                  Rule::Chain(5, TS('i'), TS('='), NS('E'), TS(';'), NS('N')),
                  Rule::Chain(4, TS('i'), TS('='), NS('E'), TS(';')),
                  Rule::Chain(8, TS('u'), TS('('), NS('E'), TS(')'), TS('{'), NS('N'),
TS('}'), NS('N')),
                  Rule::Chain(7, TS('u'), TS('('), NS('E'), TS(')'), TS('{'), NS('N'), TS('}')),
                  Rule::Chain(9, TS('d'), TS('f'), TS('t'), TS('i'), TS('('), NS('F'), TS(')'),
TS(';'), NS('N')),
                  Rule::Chain(8, TS('d'), TS('f'), TS('t'), TS('i'), TS('('), NS('F'), TS(')'),
TS(';')),
                  Rule::Chain(4, TS('p'), TS('i'), TS(';'), NS('N')),
                  Rule::Chain(4, TS('p'), TS('l'), TS(';'), NS('N')),
                  Rule::Chain(3, TS('p'), TS('i'), TS(';')),
                  Rule::Chain(3, TS('p'), TS('l'), TS(';')),
                  Rule::Chain(7, TS('p'), TS('i'), TS('('), TS('i'), TS(')'), TS(';'), NS('N')),
                  Rule::Chain(6, TS('p'), TS('i'), TS('('), TS('i'), TS(')'), TS(';'))),
            Rule(NS('E'), GRB_ERROR_SERIES + 2, // ошибка в выражении
                  7,
                  Rule::Chain(1, TS('i')),
                  Rule::Chain(1, TS('l')),
                  Rule::Chain(2, TS('i'), NS('M')),
                  Rule::Chain(3, TS('('), NS('E'), TS(')')),
                  Rule::Chain(4, TS('i'), TS('('), NS('W'), TS(')')),
                  Rule::Chain(5, TS('i'), TS('('), NS('W'), TS(')'), NS('M')),
                  Rule::Chain(2, TS('l'), NS('M'))),
            Rule(NS('M'), GRB_ERROR_SERIES + 5, // условие if
                  3,
                  Rule::Chain(2, TS('v'), NS('E')),
                  Rule::Chain(3, TS('v'), NS('E'), NS('M')),
                  Rule::Chain(2, TS('s'), NS('E'))),
            Rule(NS('F'), GRB_ERROR_SERIES + 3, // ошибка в параметрах функции
                  2,
                  Rule::Chain(4, TS('t'), TS('i'), TS(','), NS('F')),
                  Rule::Chain(2, TS('t'), TS('i'))
),
            Rule(NS('W'), GRB_ERROR_SERIES + 4, // ошибка в параметрах вызываемой функ
                  8,
                  Rule::Chain(3, TS('i'), TS(','), NS('W')),
                  Rule::Chain(3, TS('l'), TS(','), NS('W')),
                  Rule::Chain(1, TS('i')),
                  Rule::Chain(1, TS('l')),
                  Rule::Chain(4, TS('i'), NS('M'), TS(','), NS('W')),
                  Rule::Chain(4, TS('l'), NS('M'), TS(','), NS('W')),
                  Rule::Chain(2, TS('i'), NS('M')),
                  Rule::Chain(2, TS('l'), NS('M'))
            )
        );
```

Листинг 1 – Грамматика языка AVD-2022

```
	struct MfstState
	{
		short lenta_position;
		short nrule;
		short nrulechain;
		MFSTSTSTACK st;
		MfstState();
		MfstState(short pposition, MFSTSTSTACK pst, short pnrulechain);
		MfstState(short pposition, MFSTSTSTACK pst, short pnrule, short pnrulechain);
	};
	struct Mfst
	{
		enum RC_STEP { NS_OK, NS_NORULE, NS_NORULECHAIN, NS_ERROR, TS_OK, TS_NOK,
LENTA_END, SURPRISE };
		struct MfstDiagnosis
		{
			short lenta_position;
			RC_STEP rc_step;
			short nrule;
			short nrule_chain;
			MfstDiagnosis();
			MfstDiagnosis(short plenta_position, RC_STEP prc_step, short pnrule,
short pnrule_chain);
		} diagnosis[MFST_DIAGN_NUMBER];
		struct Deducation
		{
			short size;
			short* nrules;
			short* nrulechains;
			Deducation() { size = 0; nrules = 0; nrulechains = 0; };
		} deducation;
		GRBALPHABET* lenta;
		short lenta_position;
		short nrule;
		short nrulechain;
		short lenta_size;
		GRB::Greibach greibach;
		LT::LexTable lex;
		MFSTSTSTACK st;
		stack<MfstState> storestate;
		Mfst();
		Mfst(LT::LexTable plex, GRB::Greibach pgreibach);
		char* getCSt(char* buf);
		char* getCLenta(char* buf, short pos, short n = 25);
		char* getDiagnosis(short n, char* buf);
		bool savestate();
		bool reststate();
		bool push_chain(GRB::Rule::Chain chain);
		RC_STEP step();
		bool start();
		bool savediagnosis(RC_STEP pprc_step);
		bool savededucation();
		void printrules();
};
```

Листинг 2 – Структура магазинного автомата

```
struct Greibach
      {
            short size;
            GRBALPHABET startN;
            GRBALPHABET stbottomT;
            Rule* rules;
            Greibach() { short size = 0; startN = 0; stbottomT = 0, rules = 0; };
            Greibach(GRBALPHABET pstartN, GRBALPHABET pstbottomT, short psize, Rule r,
...);
            short getRule(GRBALPHABET pnn, Rule& prule);
            Rule getRule(short n);};
```

Листинг 3 – Структура грамматики Грейбах

```
0  : S->fti(F){NrE;}S    fti(ti,ti,ti){dti;i=ivivi     S$
0  : SAVESTATE:        1
1  :                   fti(ti,ti,ti){dti;i=ivivi     fti(F){NrE;}S$
2  :                   ti(ti,ti,ti){dti;i=ivivi;     ti(F){NrE;}S$
3  :                   i(ti,ti,ti){dti;i=ivivi;r     i(F){NrE;}S$
4  :                   (ti,ti,ti){dti;i=ivivi;ri     (F){NrE;}S$
5  :                   ti,ti,ti){dti;i=ivivi;ri;     F){NrE;}S$
6  : F->ti,F           ti,ti,ti){dti;i=ivivi;ri;     F){NrE;}S$
6  : SAVESTATE:        2
7  :                   ti,ti,ti){dti;i=ivivi;ri;     ti,F){NrE;}S$
8  :                   i,ti,ti){dti;i=ivivi;ri;}     i,F){NrE;}S$
9  :                   ,ti,ti){dti;i=ivivi;ri;}m     ,F){NrE;}S$
10 :                    ti,ti){dti;i=ivivi;ri;}m{     F){NrE;}S$
11 : F->ti,F           ti,ti){dti;i=ivivi;ri;}m{     F){NrE;}S$
11 : SAVESTATE:        3
12 :                   ti,ti){dti;i=ivivi;ri;}m{     ti,F){NrE;}S$
13 :                   i,ti){dti;i=ivivi;ri;}m{d     i,F){NrE;}S$
14 :                   ,ti){dti;i=ivivi;ri;}m{dt     ,F){NrE;}S$
15 :                   ti){dti;i=ivivi;ri;}m{dti     F){NrE;}S$
16 : F->ti,F           ti){dti;i=ivivi;ri;}m{dti     F){NrE;}S$
…..
855 : SAVESTATE:       76
856 :                   pl;}}                  pl;}}$
857 :                   l;}}                   l;}}$
858 :                   ;}}                    ;}}$
859 :                   }}                     }}$
860 :                   }                      }$
861 :                                          $
862 : LENTA_END
863 : ------>LENTA_END
```

Листинг 4 - Разбор исходного кода синтаксическим анализатором

```
0   : S->fti(F){NrE;}S
4   : F->ti,F
7   : F->ti,F
10  : F->ti
14  : N->dti;N
18  : N->i=E;
20  : E->iM
21  : M->vE
22  : E->iM
23  : M->vE
24  : E->i
27  : E->i
30  : S->m{N}
32  : N->dti;N
36  : N->dti;N
40  : N->dti;N
44  : N->i=E;N
46  : E->l
48  : N->i=E;N
50  : E->l
52  : N->dfti(F);N
57  : F->ti,F
60  : F->ti
64  : N->i=E;N
66  : E->i(W)
68  : W->i,W
70  : W->l
73  : N->pi;N
76  : N->dti;N
80  : N->i=E;N
82  : E->i(W)
84  : W->i,W
86  : W->i,W
88  : W->i
91  : N->pi;N
94  : N->dti;N
98  : N->i=E;N
100 : E->i(W)
102 : W->l,W
104 : W->i,W
106 : W->l
109 : N->pi;N
112 : N->dti;N
116 : N->dti;N
120 : N->i=E;N
122 : E->l
```

```
124 : N->i=E;N
126 : E->l
128 : N->dfti(F);N
133 : F->ti,F
136 : F->ti
140 : N->dti;N
144 : N->i=E;N
146 : E->i(W)
148 : W->i,W
150 : W->i
153 : N->u(E){N}N
155 : E->iM
156 : M->sE
157 : E->l
160 : N->pl;
164 : N->u(E){N}N
166 : E->iM
167 : M->sE
168 : E->i
171 : N->pl;
175 : N->u(E){N}N
177 : E->iM
178 : M->sE
179 : E->i
182 : N->pl;
186 : N->u(E){N}
188 : E->iM
189 : M->sE
190 : E->i
193 : N->pl;
```

Листинг 5 – Дерево разбора

Приложение Г

```
void SearchEq(LT::LexTable& lextable, IT::IdTable& idtable)
	{
		for (int i = 0; i < lextable.size; i++)
		{
			if ((lextable.table[i].lexema == LEX_EQUALSSIGN) ||
(lextable.table[i].lexema == LEX_RETURN) || (lextable.table[i].lexema == LEX_PRINT))
			{
				PolishNotation(i + 1, lextable, idtable);}}}
	bool PolishNotation(int lexpos, LT::LexTable& lextable, IT::IdTable& idtable)
	{
		std::stack<LT::Entry> st;
		LT::Entry outstr[200];
		int len = 0,                                          //общая длина
			lenout = 0,                                              //длина выходной ст
		semicolonid = 0;      //ид для элемента таблицы с точкой с запятой
		char t, oper = '\0';
		int hesis = 0;       //текущий символ/знак оператора/кол-во скобок
		int indoffunk;              //индекс для замены на элемент с функцией
		bool rc = false;
		int npar = 0;
		for (int i = lexpos; lextable.table[i].lexema != LEX_SEMICOLON; i++)
		{
			len = i;
			semicolonid = i + 1;
		}
		len++;
		for (int i = lexpos; i < len; i++)
		{
			t = lextable.table[i].lexema;
			if (lextable.table[i].lexema == LEX_PLUS)
				oper = idtable.table[lextable.table[i].idxTI].id[0];
			if (t == LEX_RIGHTHESIS)//выталкивание всего до другой левой скобки{
				while (st.top().lexema != LEX_LEFTHESIS)
				{
					outstr[lenout++] = st.top();//записываем в выходную строку
очередной символ между скобками
					hesis++;
					st.pop();     //удаляем вершину стека
				}
				st.pop();            //удаляем левую скобку в стеке
			}
			if (t == LEX_ID || t == LEX_LITERAL)
			{
				if (lextable.table[i + 1].lexema == LEX_LEFTHESIS)
				{
					outstr[lenout++] = lextable.table[i];
					indoffunk = i;
					i += 2;
					while (lextable.table[i].lexema != LEX_RIGHTHESIS)
					{//пока внутри аргументов функции, переписываем их в строку
						if (lextable.table[i].lexema != LEX_COMMA)
						{
							outstr[lenout++] = lextable.table[i++];
						}
						else
						{
							npar++;
							hesis++;
							i++;
						}}
					outstr[lenout++] = lextable.table[indoffunk];
					outstr[lenout - 1].lexema = LEX_NEWPROC;
					outstr[lenout - 1].par = npar + 1;
					outstr[lenout - 1].idxTI = lextable.table[indoffunk].idxTI;
					npar = 0;
					hesis += 1;}
				else
```

```
                                outstr[lenout++] = lextable.table[i];}
                        if (t == LEX_LEFTHESIS)
                        {
                                st.push(lextable.table[i]);//помещаем в стек левую скобку
                                hesis++;}

                        if (oper == '+' || oper == '-' || oper == '*' || oper == '/')
                        {
                                if (!st.size())
                                        st.push(lextable.table[i]);
                                else {
                                        int pr, id;
                                        if (st.top().lexema == '(' || st.top().lexema == ')')
                                                pr = 1;
                                        else {
                                                id = st.top().idxTI;
                                                pr = ArifmPriorities(idtable.table[id].id[0]);
                                        }
                                        if (ArifmPriorities(oper) > pr)   //если приоритет добавляемой
операции больше операции на вершине стека
                                        st.push(lextable.table[i]);//добавляем операции в стек
                                        else
                                        {
                                                while (st.size() && ArifmPriorities(oper) <=
ArifmPriorities(idtable.table[id].id[0]))//если меньше, то записываем в строку все операции с
большим или равным приоритетом
        {
                                                        outstr[lenout] = st.top();
                                                        st.pop();
                                                        lenout++;}
                                                st.push(lextable.table[i]);}}}
                        oper = NULL;                            //обнуляем поле знака
}
                while (st.size())
                {
                        outstr[lenout++] = st.top();//вывод в строку всех знаков из стека
                        st.pop();
                }
                for (int i = lexpos, k = 0; i < lexpos + lenout; i++, k++)
                {
                        lextable.table[i] = outstr[k];//запись в таблицу польской записи
                        rc = true;
                }
                lextable.table[lexpos + lenout] = lextable.table[semicolonid]; //вставка элемента
с точкой с запятой
                for (int i = 0; i < hesis; i++)
                {
                        for (int j = lexpos + lenout + 1; j < lextable.size; j++)//сдвигаем на
лишнее место
                        {
                                lextable.table[j] = lextable.table[j + 1];
                        }
                        lextable.table[lextable.size - hesis + i + 1].lexema = '-';
                        lextable.table[lextable.size - hesis + i + 1].sn = 0;
                        lextable.table[lextable.size - hesis + i + 1].idxTI = TI_NULLIDX;
                }
                return rc;
        }
        int ArifmPriorities(char symb)
        {
                if (symb == LEX_LEFTHESIS || symb == LEX_RIGHTHESIS)
                        return 1;
                if (symb == '+' || symb == '-')
                        return 2;
                if (symb == '*' || symb == '/')
                        return 3;
                                                }
```

Листинг 1 – Программная реализация механизма преобразования в ПОЛИЗ

```
01  fti(ti,ti,ti)
02  {
03  dti;
04  i=ii+i+;
05  ri;
06  }
07  m
08  {
09  dti;
10  dti;
11  dti;
12  i=l;
13  i=l;
14  dfti(ti,ti);
15  i=iil@2;
16  pi;
17  dti;
18  i=iiii@3;
19  pi;
20  dti;
21  i=ilil@3;
22  pi;
23  dti;
24  dti;
25  i=l;
26  i=l;
27  dfti(ti,ti);
28  dti;
29  i=iii@2;
30  u(isl)
31  {
32  pl;
33  }
34  u(isi)
35  {
36  pl;
37  }
38  u(isi)
39  {
40  pl;
41  }
42  u(isi)
43  {
44  pl;
45  }
46  }
```

Листинг 6 – результат преобразования к обратной польской записи

Приложение Д

```
.586P
.MODEL FLAT, stdcall
includelib libucrt.lib
includelib kernel32.lib
includelib ../Debug/StaticLib.lib
ExitProcess PROTO : DWORD
SetConsoleTitleA PROTO :DWORD
GetStdHandle PROTO :DWORD


stepen PROTO: DWORD, :DWORD
sravs PROTO: DWORD, :DWORD
printu PROTO: DWORD
prints PROTO: DWORD
su PROTO : DWORD, : DWORD, : DWORD

.STACK 8192

.CONST

	LEX1 DWORD 4
	LEX2 DWORD 2ah
	LEX3 DWORD 2
	LEX4 DWORD 1
	LEX5 DWORD 3
	LEX6 byte 'exampleExample', 0
	LEX7 byte 'example', 0
	LEX8 DWORD 1
	LEX9 byte 'Okey', 0
	LEX10 byte 'Yes', 0
	LEX11 byte 'Yes', 0
	LEX12 byte 'Yes', 0

.DATA

	ret_su DWORD ?
	sures DWORD ?
	maina DWORD ?
	mainb DWORD ?
	mainc DWORD ?
	mainsum DWORD ?
	mainsumo DWORD ?
	mainstr DWORD ?
	mainstro DWORD ?
	mainrezs DWORD ?

.CODE

su PROC a: DWORD, b: DWORD, d: DWORD
	push a
	push b
	pop eax
	pop ebx
	add eax, ebx
	push eax
	push d
	pop eax
	pop ebx
	add eax, ebx
	push eax
	pop sures
	push sures
	pop eax
ret
```

```
su ENDP

main PROC
START:
	push LEX1
	pop maina
	push LEX2
	pop mainb
	push maina
	push LEX3
	call stepen
	push eax
	pop mainc
	push mainc
	call printu
	push maina
	push mainb
	push mainc
	call su
	push eax
	pop mainsum
	push mainsum
	call printu
	push LEX4
	push maina
	push LEX5
	call su
	push eax
	pop mainsumo
	push mainsumo
	call printu
	push offset LEX6
	pop mainstr
	push offset LEX7
	pop mainstro
	push mainstr
	push mainstro
	call sravs
	push eax
	pop mainrezs
mov eax, mainrezs
cmp eax, LEX8
je equal0
jne nequal0
equal0:
	push offset LEX9
	call prints
nequal0:
mov eax, mainb
cmp eax, maina
jne nequal1
je equal1
nequal1:
	push offset LEX10
	call prints
equal1:
mov eax, maina
cmp eax, mainc
jae more2
jb less2
more2:
	push offset LEX11
	call prints
less2:
mov eax, mainb
cmp eax, mainc
jbe less3
```

```
ja more3
less3:
       push offset LEX12
       call prints
more3:
call ExitProcess
main ENDP

end main
```

Листинг 1 – Сгенерированный код на языке Assembler

```
16
62
8
Yes
```

Листинг 2 – Результат работы файла на языке Assemble