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 TTM-2020»

Выполнил студент Тихомиров Тимофей Михайлович
(Ф.И.О.)

Руководитель проекта пр.ст. Пахолко Алена Степановна
(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 пр.ст. Пахолко Алена Степановна
(учен. степень, звание, должность, подпись, Ф.И.О.)

Нормоконтролер пр.ст. Пахолко Алена Степановна
(учен. степень, звание, должность, подпись, Ф.И.О.)

Курсовой проект защищен с оценкой

Минск 2020

Содержание

Введение

Целью курсового проекта поставлена задача разработки компилятора для моего языка программирования ТТМ-2020. Этот язык программирования предназначен для выполнения простейших операций и арифметических действий над числами.

Компилятор ТТМ-2020 – это программа, задачей которого является перевод программы, написанной на языке программирования ТТМ-2020 в программу на язык ассемблера.

Транслятор ТТМ-2020 состоит из следующих частей:

– лексический и семантический анализаторы;

– синтаксический анализатор;

– генератор исходного кода на языке ассемблера.

Исходя из цели курсового проекта, были определены следующие задачи:

– разбработка спецификации языка программирования;

– разбратка структуры транслятора;

– разработка лексического и семантического анализаторов;

– разработка синтаксического анализатора;

– преобразование выражений;

– генерация кода на язык ассемблера;

– тестирование транслятора.

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

Глава 1. Спецификация языка программирования

1.1 Характеристика языка программирования

Язык программирования TTM-2020 классифицируется как процедурный, универсальный, строготипизированный, компилируемый и не объектно-ориентированный язык.

1.2 Алфавит языка

Алфавит языка TTM-2020 основан на кодировке Windows-1251, представленной 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

На этапе выполнения могут использоваться символы латинского алфавита, цифры десятичной системы счисления от 0 до 9, спецсимволы, а также непечатные символы пробела, табуляции и перевода строки.

1.3 Символы сепараторы

Символы, которые являются сепараторами представлены в таблице 1.1.

Таблица 1.1 – Сепараторы

Сепаратор	Название	Область применения
‘ ‘	Пробел	Допускается везде, кроме идентификаторов и ключевых слов
;	Точка с запятой	Разделение конструкций
{…}	Фигурные скобки	Заключение программного блока и блока if или else
(…)	Круглые скобки	Приоритет операций, параметры функции
‘…’	Одинарные кавычки	Допускается везде, кроме идентификаторов и ключевых слов
=	Знак «равно»	Присваивание значения
,	Запятая	Разделение параметров
+ - * / %	Знаки «плюс», «минус», «умножить», «разделить», «остаток от деления»	Выражения

1.4 Применяемые кодировки

Для написания исходного кода на языке программирования TTM-2020 используется кодировка Windows-1251.

1.5 Типы данных

В языке TTM-2020 реализованы два типа данных: целочисленный и строковый. Описание реализованных типов данных представлено в таблице 1.2.

Таблица 1.2 – Типы данных языка TTM-2020

Тип данных	Описание типа данных
Целочисленный тип данных i32	Фундаментальный тип данных. Используется для работы с целочисленными значениями. В памяти занимает 4 байта. Максимальное значение: 2,147,483,647. Минимальное значение: -2,147,483,648. Инициализация по умолчанию: значение 0.
Строковый тип данных str	Фундаментальный тип данных. Используется для работы с символами, каждый символ в памяти занимает 1 байт. Максимальное количество символов: 255. Инициализация по умолчанию: длина 0, символ конца строки “\0”.

1.6 Преобразование типов данных

Преобразование типов данных не поддерживается, т.е. язык является строго-типизированным.

1.7 Идентификаторы

В имени идентификатора допускаются символы латинского алфавита в нижнем и верхнем регистре. Имя идентификатора не может совпадать с ключевыми словами и не может иметь имя, как функция, уже содержащаяся в стандартной библиотеке.

1.8 Литералы

С помощью литералов осуществляется инициализация переменных. В языке существует два типа литералов. Краткое описание литералов языка ТТМ-2020 представлено в таблице 1.3.

Таблица 1.3 – Описание литералов

Тип литерала	Регулярное выражение	Описание	Пример
Целочисленный литерал	[1-9]+[0-9]*	Целочисленные литералы, по умолчанию инициализируются 0. Литералы могут быть только rvalue.	let i32 sum; sum = 15; 15 – целочисленный литерал.
Строковые литерал	[a-z|A-Z|A-Я|a-я|0-9|!-/]+	Символы, заключённые в '…' (двойные кавычки), по умолчанию инициализируются пустой строкой. Литералы могут быть только rvalue.	let str text = ‘text’; text – строковый литерал.

1.9 Область видимости идентификаторов

Область видимости «сверху вниз» (по принципу C++). В языке ТТМ-2020 требуется обязательное объявление переменной перед её инициализацией и последующим использованием. Все переменные должны находиться внутри программного блока. Имеется возможность объявления одинаковых переменных в разных блоках, т. к. переменные, объявленные в одной функции, недоступны в другой. Каждая переменная получает префикс – название функции, в которой она объявлена.

1.10 Инициализация данных

При объявлении переменной допускается инициализация данных. Краткое описание способов инициализации переменных языка ТТМ-2020 представлено в таблице 1.4.

Таблица 1.4 – Способы инициализации переменных

Конструкция	Описание	Пример
let <тип данных> <идентификатор>;	Автоматическая инициализация: переменные типа i32 инициализируются нулём, переменные типа str – пустой строкой.	let i32 sum; let str text;
<идентификатор> = <значение>;	Присваивание переменной значения.	sum = 7; text = ‘text’;
let <тип данных> <идентификатор> = <значение>;	Присваивание переменной значения.	let i32 sum = 228; let str text = ‘text’

1.11 Инструкции языка

Все возможные инструкции языка программирования ТТМ-2020 представлены в общем виде в таблице 1.5.

Таблица 1.5 – Инструкции языка программирования ТТМ -2020

Инструкция	Запись на языке ТТМ -2020
Объявление переменной	let <тип данных> <идентификатор>;
Присваивание	<идентификатор> = <значение>/<идентификатор>;
Объявление внешней функции	fn <тип данных> <идентификатор> (<тип данных> <идентификатор>, …)

Блок инструкций	fn i32 main()
	{
	…
	}

Возврат из подпрограммы	ret <идентификатор> / <литерал>;
Условная инструкция	if <идентификатор> / <литерал> {/программный блок если условие верно/} else {/программный блок если условие ложно/}
Вывод данных	echo <идентификатор> / <литерал>;

1.12 Операции языка

Язык программирования ТТМ-2020 может выполнять операции сравнения, представленные в таблице 1.6.

Таблица 1.6 – Операции языка программирования ТТМ-2020

Операция	Примечание	Типы данных	Пример
=	Присваивание	(i32, i32) (str, str)	sum = 7; text = “text”;
+	Знак «плюс»	(i32, i32)	sum + diff
-	Знак «минус»	(i32, i32)	sum - diff
(	Приоритет операций	-	sum = (a + b) - c;
)			
*	Знак «умножить»	(i32, i32)	a * b;
/	Знак «разделить»	(i32, i32)	a / b;
%	Знак «остаток от деления»	(i32, i32)	a % b;

1.13 Выражения и их вычисления

Круглые скобки в выражении используются для изменения приоритета операций. Выражение может содержать вызов функции, если эта функция уже содержится в стандартной библиотеке.

1.14 Программные конструкции языка

Ключевые программные конструкции языка программирования ТТМ-2020 представлены в таблице 1.7.

Таблица 1.7 – Программные конструкции языка ТТМ-2020

Конструкция	Запись на языке ТТМ-2020

Главная функция (точка входа)	fn i32 main()
	{
	…
	ret <идентификатор> / <литерал>;
	}

Функция	fn <тип> <идентификатор> (<тип> <идентификатор>, …)
	{
	…
	ret <идентификатор> / <литерал>;
	}

1.15 Область видимости

В языке ТТМ-2020 все переменные являются локальными. Они обязаны находится внутри программного блока функций (по принципу C++). Объявление глобальных переменных не предусмотрено. Объявление пользовательских областей видимости не предусмотрено.

1.16 Семантические проверки

Таблица с перечнем семантических проверок, предусмотренных языком, приведена в таблице 1.8.

Таблица 1.8 – Семантические проверки

Номер	Правило
1	Идентификаторы не должны повторяться
2	Тип возвращаемого значения должен совпадать с типом функции при её объявлении
3	Тип данных передаваемых значений в функцию должен совпадать с типом параметров при её объявлении
4	В функцию должны быть переданы параметры
5	Тип данных результата выражения должен совпадать с типом данных идентификатора, которому оно присваивается

1.17 Распределение оперативной памяти на этапе выполнения

Все переменные размещаются в стеке.

1.18 Стандартная библиотека и её состав

Стандартная библиотека ТТМ-2020 написана на языке программирования C++. Функции стандартной библиотеки с описанием представлены в таблице 1.9.

Таблица 1.9 – Состав стандартной библиотеки

Функция(C++)	Возвращаемое значение	Описание
const char* __stdcall _concat(const char* str1, const char* str2);	i32	Конкатенация строк
int __stdcall _parseInt(const char* str);	i32	Функция конвертации строки в целочисленное значение
void __stdcall _echoStr(const char* str)		Функция вывода на консоль строкового идентификатора/литерала
void __stdcall _echoInt(int num)		Функция вывода на консоль целочисленного идентификатора/литерала

1.19 Ввод и вывод данных

В языке ТТМ-2020 не реализованы средства ввода данных.

Для вывода данных в стандартный поток вывода предусмотрен оператор echo, который входят в состав стандартной библиотеки и описан в таблице 1.9.

1.20 Точка входа

В языке ТТМ-2020 каждая программа должна содержать главную функцию main, т. е. точку входа, с которой начнется последовательное выполнение программы.

1.21 Препроцессор

Препроцессор в языке программирования ТТМ-2020 не предусмотрен.

1.22 Соглашения о вызовах

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

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

1.23 Объектный код

ТТМ-2020 транслируется в язык ассемблера.

1.24 Классификация сообщений транслятора

В случае возникновения ошибки в коде программы на языке ТТМ-2020 и выявления её транслятором в текущий файл протокола выводится сообщение. Их классификация сообщений приведена в таблице 1.10.

Таблица 1.10. Классификация сообщений транслятора

Интервал	Описание ошибок
0-99	Системные ошибки
100-109	Ошибки параметров
110-119	Ошибки открытия и чтения файлов
120-140	Ошибки лексического анализа
600-610	Ошибки синтаксического анализа
700-710	Ошибки семантического анализа

Глава 2. Структура транслятора

2.1 Компоненты транслятора, их назначение и принципы взаимодействия

Основными компонентами транслятора являются лексический анализатор, синтаксический анализатор, семантический анализатор и генератор кода, приведенные на рисунке 2.1.

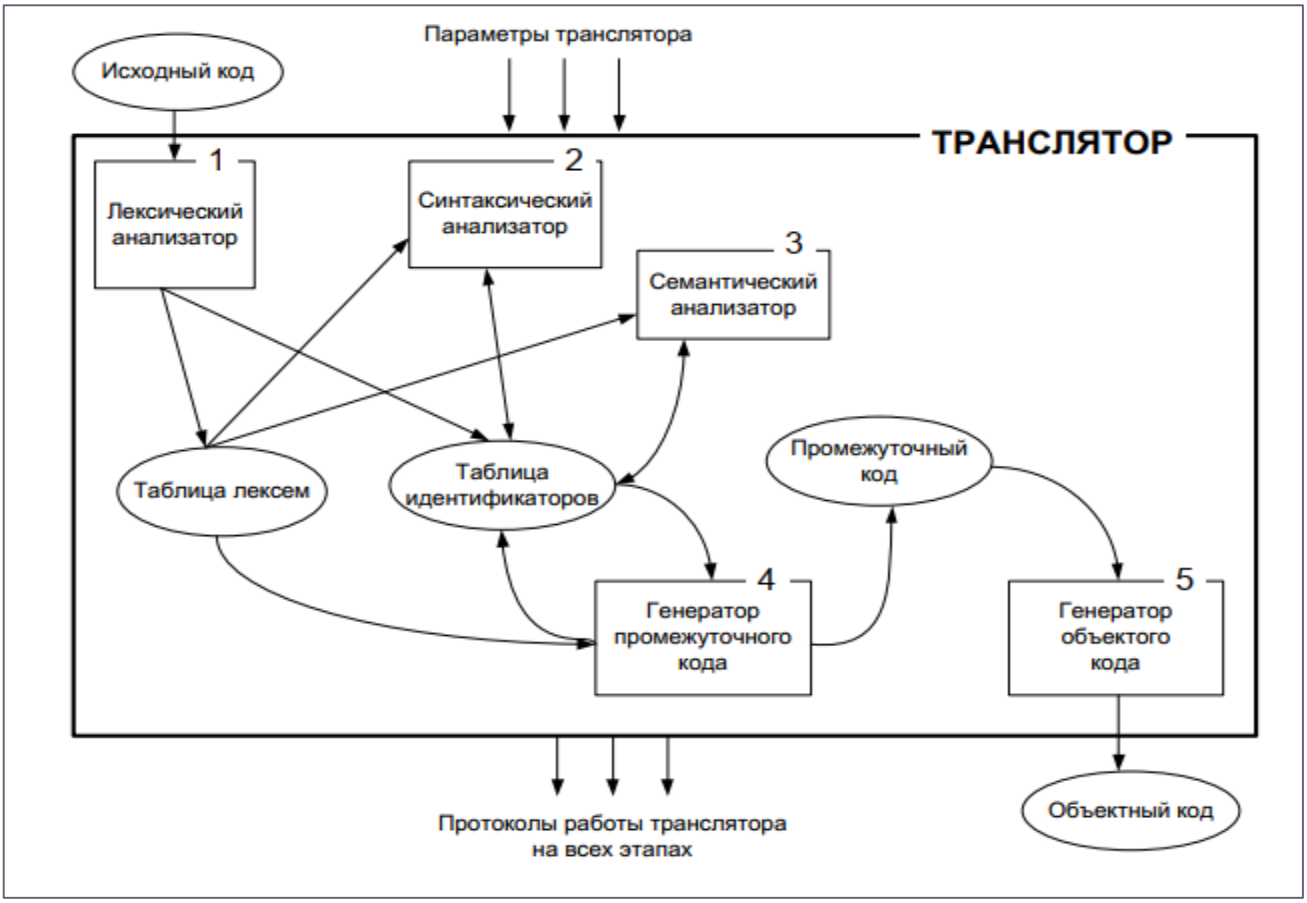

Рисунок 2.1 – Структура транслятора

Транслятор – это программа преобразующая исходный код на одном языке программирования в исходный код на другом языке программирования.

Лексический анализатор – принимает на вход уже первично обработанный и разбитый на отдельные компоненты исходный код на языке ТТМ-2020. Формирует таблицу идентификаторов и таблицу лексем, а также занимается обнаружением ошибок, связанных с лексикой языка.

Синтаксический анализатор – принимает на вход таблицу лексем, сформированную лексическим анализатором. Перебирая каждое правило языка (допустимую конструкцию) он выявляет синтаксические ошибки, допущенные в исходном коде. Формирует дерево разбора, а также выводит трассировку (разбор) цепочек.

Семантический анализатор – состоит из нескольких функций, отвечающих за выявления тех или иных ошибок, а также некоторых проверок, выполняемых на этапе лексического анализатора. В зависимости от задачи функции на ее вход подается таблица лексем либо таблица лексем и идентификаторов.

Генератор кода – принимает на вход таблицу идентификаторов и таблицу лексем. Задача этого компонента заключается в трансляции, уже пройденного все предыдущие этапы кода на языке ТТМ-2020, в код на языке Ассемблер.

2.2 Перечень входных параметров транслятора

Входные параметры транслятора представлены в таблице 2.1.

Таблица 2.1 – Входные параметры транслятора языка ТТМ-2020

Входной параметр	Описание	Значение по умолчанию
-in:<имя_файла>	Входной файл с расширением .ttm, в котором содержится исходный код языка ТТМ-2020	Не предусмотрено
-log:<имя_файла>	Файл, содержащий информацию о работе транслятора.	<имя_файла>.log
-asm:<имя_файла>	Файл для записи результата работы лексического и синтаксического анализаторов.	<имя_файла>.asm

2.3 Перечень протоколов, формируемых транслятором и их содержимое

Протоколы транслятора ТТМ-2020 представлены в таблице 2.2.

Таблица 2.2 – Протоколы транслятора ТТМ-2020

Протокол	Содержимое
<имя_файла>.log	Содержит служебную информацию, получаемую в ходе работы транслятора, а так же ошибки, которые возникают на этапе обработки исходного кода.

Протокол работы нужен для отображения хода выполнения трансляции языка ТТМ-2020. Благодаря им пользователь может обнаружить некорректно введенные данные или ошибки в исходном коде программы.

Глава 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 последовательность символов входного языка. Лексический анализатор выделяет в этой последовательности простейшие конструкции языка, которые называют лексическими единицами. 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. Результатом работы лексического анализатора являются таблица лексем и идентификаторов. Структура лексического анализатора представлена на рисунке 3.1.

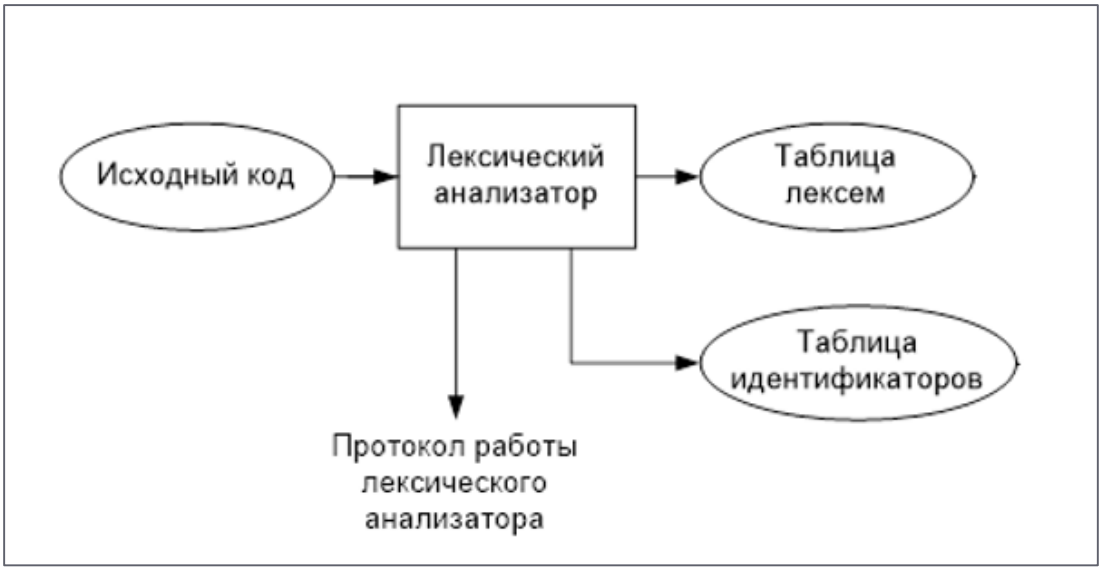

Рисунок 3.1 – Структура лексического анализатора ТТМ-2020

3.2 Контроль входных символов

Принцип работы таблицы заключается в соответствии значения каждому элементу в шестнадцатеричной системе счисления значению в таблице ASCII.

На рисунке 3.2 представлена таблица входных символов.

```
#define IN_CODE_TABLE \
        in::F, in::F, in::F, in::F, in::F, in::F, in::F, in::F, in::F, in::delimiter,in::delimiter, in::F, in::F, in::I, in::F, in::F, \
        in::F, in::F, in::F, in::F, in::F, in::F, in::F, in::F, in::F, in::F, in::F, in::F, in::F, in::F, in::F, in::F, \
        in::delimiter, in::F, in::F, in::F, in::F, in::O, in::F, in::Q, in::S, in::S, in::O, in::O, in::S, in::O, in::F, in::O, \
        in::T, in::T, in::T, in::T, in::T, in::T, in::T, in::T, in::T, in::T, in::F, in::S, in::F, in::O, 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::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::S, in::F, in::S, in::F, in::F, \

        in::F, in::F, in::F, in::F, in::F, in::F, in::F, in::F, in::F, in::F, in::F, in::F, in::F, in::F, in::F, in::F, \
        in::F, in::F, in::F, in::F, in::F, in::F, in::F, in::F, in::F, in::F, in::F, in::F, in::F, in::F, in::F, in::F, \
        in::F, in::F, in::F, in::F, in::F, in::F, in::F, in::F, in::F, in::F, in::F, in::F, in::F, in::F, in::F, in::F, \
        in::F, in::F, in::F, in::F, in::F, in::F, in::F, in::F, in::F, in::F, in::F, in::F, in::F, in::F, in::F, in::F, \
        in::F, in::F, in::F, in::F, in::F, in::F, in::F, in::F, in::F, in::F, in::F, in::F, in::F, in::F, in::F, in::F, \
        in::F, in::F, in::F, in::F, in::F, in::F, in::F, in::F, in::F, in::F, in::F, in::F, in::F, in::F, in::F, in::F, \
        in::F, in::F, in::F, in::F, in::F, in::F, in::F, in::F, in::F, in::F, in::F, in::F, in::F, in::F, in::F, in::F, \
        in::F, in::F, in::F, in::F, in::F, in::F, in::F, in::F, in::F, in::F, in::F, in::F, in::F, in::F, in::F, in::F
```

Рисунок 3.2. – Таблица контроля входных символов

Описание значения символов: T – разрешённый символ, F – запрещённый символ, I – игнорируемый символ, O – операция, S – сепаратор, Q – кавычка, delimiter – разделитель лексем.

3.3 Удаление избыточных символов

Избыточными символами являются символы табуляции и пробелы.

Избыточные символы удаляются на этапе разбиения исходного кода на лексемы.

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

Лексемы – это символы, соответствующие ключевым словам, символам операций и сепараторам, необходимые для упрощения дальнейшей обработки исходного кода программы. Соответствие цепочек и лексем представлено в таблице 3.1.

Таблице 3.1 – Таблица соответствий цепочек, их типов и лексем

Цепочка	Тип	Лексема
i32	Ключевое слово	n
str	Ключевое слово	s
Идентификатор	Переменная, функция или параметр	i
Целочисленный литерал	Данные	1
Строковый литерал	Данные	2
main	Ключевое слово	m
fn	Ключевое слово	f
let	Ключевое слово	d
echo	Ключевое слово	p

Окончание таблицы 3.1

Цепочка	Тип	Лексема
if	Ключевое слово	I
else	Ключевое слово	E
+ - * / = %	Операторы	+ - * / = %
; , {} ()	Сепараторы	; , {} ()

Пример реализации таблицы лексем представлен в приложении А.

Также в приложении А находятся конечные автоматы, соответствующие лексемам языка ТТМ-2020.

3.5 Основные структуры данных

Основные структуры таблиц лексем и идентификаторов данных языка ТТМ-2020, используемых для хранения, представлены в приложении А. В таблице лексем содержится лексема, её номер, полученный при разборе, номер строки в исходном код. В таблице идентификаторов содержится имя идентификатора, номер в таблице лексем, тип данных, смысловой тип идентификатора и его значение.

3.6 Принцип обработки ошибок

В случае если в результате работы лексического анализатора найдена ошибка, то анализатор заполняет структуру, содержащую ошибки, произошедшие в процессе лексического анализа.

После чего вызывается функция генерации ошибки, в которую передается, в зависимости от места возникновения ошибки, следующая информация: код ошибки, номер строки в коде, номер позиции в строке или только код ошибки. Пользователь может ознакомиться с данной ошибкой, открыв протокол.

3.7 Структура и перечень сообщений лексического анализатора

Сообщение лексического анализатора представлено на рисунке 3.3.

```
ERROR_ENTRY(123, "Повторное объявление лексемы")
```

Рисунок 3.3 – Ошибка лексического анализатора

3.8 Параметры лексического анализатора и режимы его работы

Входным параметром лексического анализа является очередь, состоящая из структур, полями которых являются лексема и номер её строки в исходном файле, полученные на этапе проверки исходного кода на допустимость символов.

3.9 Алгоритм лексического анализа

Лексический анализатор проверяет входной поток символов программы на исходном коде на допустимость, удаляет лишние пробелы. 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 началу. При неуспешном распознавании выдается сообщение об ошибке, а дальнейшие действия зависят от реализации анализатора – либо его выполнение прекращается, либо делается попытка распознать следующую лексему(идет возврат к началу алгоритма).

3.10 Контрольный пример

Результат работы лексического анализатора – таблицы лексем и идентификаторов – представлен в приложении А.

Глава 4. Разработка синтаксического анализатора

4.1 Структура синтаксического анализатора

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

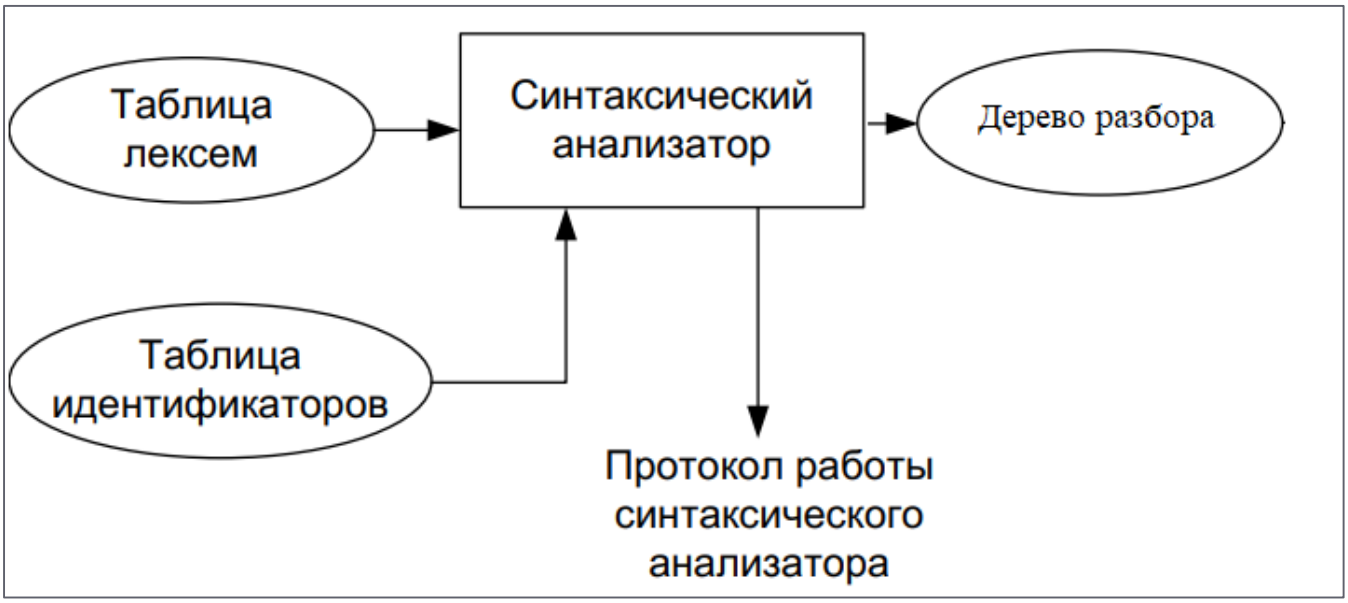

Рисунок 4.1 – Структура синтаксического анализатора

4.2 Контекстно-свободная грамматика, описывающая синтаксис языка

В синтаксическом анализаторе транслятора языка ТТМ-2020 используется контекстно-свободная грамматика $G = \langle T, N, P, S \rangle$, где

T – множество терминальных символов,
N – множество нетерминальных символов (первый столбец таблицы 4.1),
P – множество правил языка (второй столбец таблицы 4.1),
S – начальный символ грамматики, являющийся нетерминалом.

Эта грамматика имеет нормальную форму 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.

Грамматика языка ТТМ-2020 представлена в приложении Б.

TS – терминальные символы, NS – нетерминальные символы.

Таблица 4.1 – Перечень правил, составляющих грамматику языка и описание нетерминальных символов ТТМ-2020

Нетерминал	Цепочки правил	Описание
S	fti(F){NrE;}S ftm(){NrE;}	Правила для общих структур
N	dti;N i=E;N; dti=E;N pi;N pl;N IE{N}N IE{N}E{N}N	Правила для конструкций и инструкций
E	i l (E) i(W) iM lM (E)M i(W)M i(W)	Правила для выражений
M	vE v(E) v(E)M vEM	Правила для операторов
F	ti ti,F	Правила для параметров функции
W	i l i,W l,W	Правила для параметров вызываемой функции

4.3 Построение конечного магазинного автомата

В данном курсовом проекте грамматика приведена к нормальной форме Грейбах. Это означает, что каждое правило имеет вид A → aα, где a∈T, α∈N. Конечный автомат с магазинной памятью можно представить в виде семёрки $M = \langle Q, V, Z, \delta, q_0, z_0, F \rangle$, где M – автомат, Q – множество состояний, V – алфавит входных символов, Z – алфавит магазина, δ - функция переходов, q_0 – начальное состояние автомата, z_0 – начальное состояние магазина, F – множество конечных состояний.

4.4 Основные структуры данных

Основные структуры данных синтаксического анализатора включают в себя структуру магазинного автомата и структуру грамматики Грейбах, описывающей правила языка ТТМ-2020. Данные структуры представлены в приложении Б.

4.5 Описание алгоритма синтаксического разбора

Алгоритм синтаксического разбора:

1) Поиск и выделение синтаксических конструкций в исходном тексте.
2) Распознавание (проверка правильности) синтаксических конструкций.
3) Выявление ошибок и продолжение процесса распознавания после обработки ошибок.
4) Если нет ошибок, формирование дерева разбора.

4.6 Структура и перечень сообщений синтаксического анализатора

Перечень сообщений синтаксического анализатора представлен на рисунке 4.3.

```
ERROR_ENTRY(600,"Неверная структура программы"),
ERROR_ENTRY(601,"Ошибочный оператор"),
ERROR_ENTRY(602,"Ошибка в выражении"),
ERROR_ENTRY(603,"Ошибка в подвыражении"),
ERROR_ENTRY(604,"Ошибка в параметрах функции"),
ERROR_ENTRY(605,"Ошибка в параметрах вызываемой функции"),
ERROR_ENTRY_NODEF(606), ERROR_ENTRY_NODEF(607), ERROR_ENTRY_NODEF(608),
ERROR_ENTRY(609, "Неверный номер правила"),
```

Рисунок 4.3 – 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.

Выходным параметром является дерево разбора, которое записывается в выходной файл.

4.8 Принцип обработки ошибок

Синтаксический анализатор перебирает всевозможные правила грамматики для нахождения подходящего соответствия с конструкцией, представленной в таблице лексем.

Если не была найдена ни одна подходящая цепочка, то формируется соответствующая ошибка. Пользователь может ознакомиться с данной ошибкой, открыв протокол.

4.9 Контрольный пример

Пример разбора синтаксическим анализатором исходного кода на языке ТТМ-2020 представлен в приложении Б. Дерево разбора исходного кода также представлено в приложении Б.

Глава 5. Разработка семантического анализатора

5.1 Структура семантического анализатора

Семантический анализ происходит после выполнения синтаксического анализа и реализуется в виде отдельных проверок таблиц литералов и идентификаторов на соответствие правилам.

5.2 Функции семантического анализатора

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

5.3 Структура и перечень сообщений семантического анализатора

Сообщения, формируемые семантическим анализатором, представлены на рисунке 5.1.

```
ERROR_ENTRY(700, "Ошибка в возвращаемом значении"),
ERROR_ENTRY(701, "Ошибка в параметрах функции"),
ERROR_ENTRY_NODEF(702),
ERROR_ENTRY_NODEF(703),
ERROR_ENTRY_NODEF(704),
ERROR_ENTRY_NODEF(705),
ERROR_ENTRY(706, "Несоответсвие присваимаего типа данных"),
ERROR_ENTRY(707, "Неверные типы данных операндов"),
ERROR_ENTRY_NODEF(708),
ERROR_ENTRY_NODEF(709),
ERROR_ENTRY(710, "Ошибка в условной конструкции"),
```

Рисунок 5.1 – Перечень сообщений семантического анализатора

5.4 Принцип обработки ошибок

В качестве входных параметров выступают таблица литералов и идентификаторов. Далее происходит полный анализ данных таблиц. В случае возникновения ошибок, вызываем функцию получения ошибки, которая принимает обязательным параметром код ошибки в таблице сообщений, номер строки и позицию. Пользователь может ознакомиться с данной ошибкой, открыв протокол.

5.5 Контрольный пример

Результат работы контрольного примера расположен в приложении В.

Глава 6. Преобразование выражений

6.1 Выражения, допускаемые языком

Выражения и операции, допускаемые языком, подробно описаны в разделе 1.12 и 1.13.

6. Выражения в языке TAV-2020 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:

- читаем очередной символ;
- если он является идентификатором или литералом, то добавляем его к выходной строке;
- если символ является символом функции, то помещаем его в стек;
- если символ является открывающей скобкой, то она помещается в стек;
- исходная строка просматривается слева направо;
- если символ является закрывающей скобкой, то выталкиваем из стека в выходную строку все символы пока не встретим открывающую скобку. При этом обе скобки удаляются и не попадают в выходную строку;
- как только входная лента закончится все символы из стека выталкиваются в выходную строку;
- в случае если встречаются операции, то выталкиваем из стека в выходную строку все операции, которые имеют выше приоритетность чем последняя операция;
- также, если идентификатор является именем функции, то он заменяется на спецсимвол «@».

Таблица 6.2 – Пример преобразования выражения в обратную польскую запись

Исходная строка	Результирующая строка	Стек
x+y*5/(z-2)		
+y*5/(z-2)	x	
y*5/(z-2)	x	+
*5/(z-2)	xy	+
5/(z-2)	xy	+*
/(z-2)	xy5	+*
(z-2)	xy5*	+/
z-2)	xy5*	+/(
-2)	xy5*z	+/(
2)	xy5*z	+/(-

Окончание таблицы 6.2

Исходная строка	Результирующая строка	Стек
)	xy5*z2	+/(-
	xy5*z2-	+/
	xy5*z2-/	+
	xy5*z2-/+	

Как результат успешного разбора, мы получаем пустой стек и заполненную результирующую строку.

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 приведены изменённая таблица лексем, отображающая результаты преобразования выражений в польский формат.

Глава 7. Генерация кода

7.1 Структура генератора кода

Генерация кода – заключительный этап работы транслятора. Результатом данного этапа будет код, сгенерированный для выполнения на Ассемблере на основе таблицы лексем и таблицы идентификаторов, что является требуемым результатом работы программы. Транслятор кода начинает свою работу только в том случае если код на языке ТТМ-2020 прошёл предыдущие компоненты транслятора без ошибок. Схема данного этапа изображена на Рисунке 7.1.

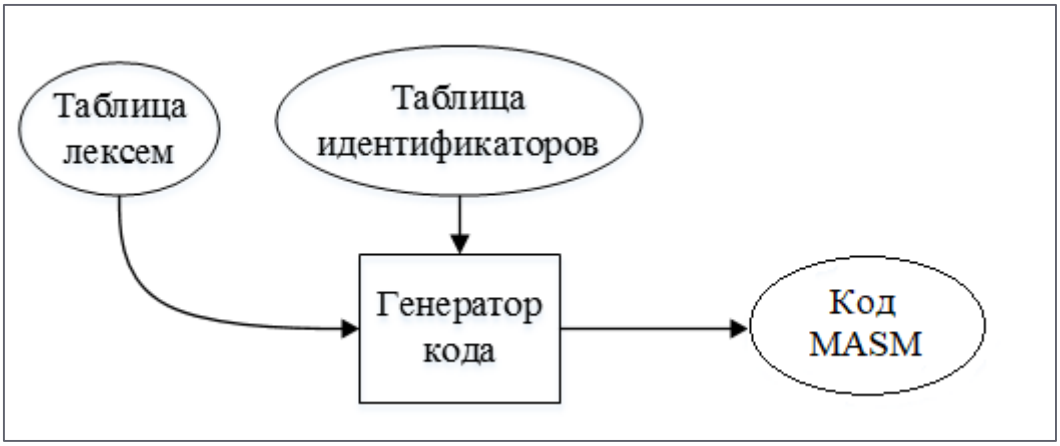

Рисунок 7.1 – Структура генератора кода

7.2 Представление типов данных в оперативной памяти

Элементы таблицы идентификаторов расположены в разных сегментах языка ассемблера – .data и .const. Идентификаторы языка ТТМ-2020 размещены в сегменте данных(.data). Литералы – в сегменте констант (.const). Соответствия между типами данных идентификаторов на языке ТТМ-2020 и на языке ассемблера приведены в таблице 7.1.

Таблица 7.1 – Соответствия типов идентификаторов языка ТТМ-2020 и языка Ассемблера

Тип идентификатора на языке ТТМ-2020	Тип идентификатора на языке ассемблера	Пояснение
i32	SDWORD	Хранит целочисленный тип данных без знака (4 байта).
str	SDWORD	Хранит строку (1 байт).

7.3 Алгоритм работы генератора кода

Генерация кода происходит на основе таблицы лексем и идентификаторов. Каждый сегмент кода Ассемблера описывается отдельно(.const, .data, .code).

В сегменте .const описываются литералы, в .data – идентификаторы, а в .code описываются конструкции. Сегмент .code разделен на 2 части: описание главной функции и локальных функций. Каждая строка кода описывается блоками. Результат преобразования записывается в файл с расширением .asm, который расположен в готовом проекте.

7.4 Состав статической библиотеки

Все функции статической библиотеки описаны в разделе 1.18.

7.5 Контрольный пример

Результат работы генерации кода представлен в приложении Д.

Глава 8. Тестирование транслятора

8.1 Тестирование фазы проверки на допустимость символов

В языке ТТМ-2020 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
x = .	Ошибка 111:Недопустимый символ в исходном коде (-in), строка 1, позиция 3.
x = я	Ошибка 111:Недопустимый символ в исходном коде (-in), строка 2, позиция 2.

8.2 Тестирование лексического анализатора

На этапе лексического анализа могут возникнуть ошибка, описанная в пункте 3.7. Результаты тестирование лексического анализатора показано в таблице 8.2.

Таблица 8.2 – Тестирование лексического анализатора

Исходный код	Диагностическое сообщение
let str X; let str X;	Ошибка 123: Повторное объявление лексемы
let str №;	Ошибка 129: Неопознанная лексема

fn i32 main()
{
 	
};
fn i32 main()
{
 	
};
	Ошибка 131: Несколько точек входа

8.3 Тестирование синтаксического анализатора

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

Таблица 8.3 – Тестирование синтаксического анализатора

Исходный код	Диагностическое сообщение
let str x =;	Ошибка 600: Неверная структура программы

8.4 Тестирование семантического анализатора

Итоги тестирования семантического анализатора приведены в таблице 8.4.

Таблица 8.4 – Тестирование семантического анализатора

Исходный код	Диагностическое сообщение
let i32 x; x = ‘f’;	Ошибка 705: Несоответствие присваиваемого типа данных

main
{
ret ‘2’;
};
	Ошибка 700: Ошибка в возвращаемом значении

i32 fn f(i32 a)
{
ret a;
};
 	
main
{
let i32 num;
num = f(‘2’);
ret 0;
};
	Ошибка 701: Ошибка в параметрах функции

Приложение А
Контрольный пример

```
//контрольный пример
fn i32 findSum( i32 x , i32 y)
{
ret x+y;
}

fn str msg(str s)
{
let str message = concat(s,s);
ret concat(message,s);
}

fn i32 main()
{
let i32 x = parseInt(‘2’);
let i32 sum;
sum = findSum(x,3);
if sum {
echo sum;
}
else {
echo ‘ложь’;
}

let str message = msg(‘hello ’);
if message {
echo message;
}
else {
echo ‘empty’;
}
ret 0;
}
```

Структуры данных лексического анализатора

```
namespace TTM
{
	class LexTable
	{
	public:
		struct Entry
		{
			char lexeme;
			int lineNumber;
			int idTableIndex;

			Entry(char lexeme, int lineNumber, int idTableIndex =
TI_NULLIDX);
		};

		LexTable(size_t capacity = 0);
		void addEntry(const LexTable::Entry& entry);
		const std::string dumpTable(size_t startIndex = 0, size_t endIn-
dex = 0) const;

		bool declaredFunction() const
		{
			return m_table.size() >= 2 && m_table[m_table.size() -
2].lexeme == LEX_FN;
		}

		bool declaredVariable() const
		{
			return m_table.size() >= 2 && m_table[m_table.size() -
2].lexeme == LEX_LET;
		}

		bool declaredDatatype() const
		{
			return m_table.size() >= 1 && (m_table[m_table.size() -
1].lexeme == LEX_DATATYPE);
		}

		int size() const { return m_table.size(); }

		bool hasLexeme(char lexeme) const;

		int lexemeCount(char lexeme) const;

		Entry& operator[](size_t index)
		{
			return m_table[index];
		}
```

```
            const Entry& operator[](size_t index) const
            {
                  return m_table[index];
            }

      private:
            std::vector<Entry> m_table;
      };
}
```

```
namespace TTM
{
	namespace it
	{
		enum class data_type { i32, str, undefined };
		enum class id_type { variable, function, parameter, literal, un-
known };
	}

	class IdTable
	{
	public:
		struct Entry
		{
			std::string name;
			std::string scope;
			int lexTableIndex;
			it::data_type dataType;
			it::id_type idType;
			union
			{
				int intValue;
				struct
				{
					char length;
					char string[TI_STR_MAXSIZE - 1];
				} strValue;
			} value;

			void setValue(int new_value);
			void setValue(const char* new_value);

			Entry() = default;
			Entry(std::string name, std::string scope, int lexTableIn-
dex, it::id_type idType, int value);
			Entry(std::string name, std::string scope, int lexTableIn-
dex, it::id_type idType, const char* value);
			Entry(std::string name, std::string scope, int lexTableIn-
dex, it::data_type dataType, it::id_type idType, const char* value);
		};

		int getIdIndexByName(std::string scope, std::string name);
		int getLiteralIndexByValue(int value);
		int getLiteralIndexByValue(const char* value);

		int addEntry(const Entry& entry);

		int size() const { return m_table.size(); }
```

```
            const std::string dumpTable(size_t startIndex = 0, size_t endIn-
dex = 0) const;

            Entry& operator[](size_t index)
            {
                  return m_table[index];
            }

            const Entry& operator[](size_t index) const
            {
                  return m_table[index];
            }

      private:
            std::vector<Entry> m_table;
      };
}
```

Таблица лексем

index	lexeme	idTable index
0	'f'	
1	't'	
2	'i'	5
3	'('	
4	't'	
5	'i'	6
6	','	
7	't'	
8	'i'	7
9	')'	
10	'{'	
11	'r'	
12	'i'	6
13	'i'	7
14	'+'	
15	';'	
16	'}'	
17	'f'	
18	't'	
19	'i'	8
20	'('	
21	't'	
22	'i'	9
23	')'	
24	'{'	
25	'd'	
26	't'	
27	'i'	10
28	'='	
29	'i'	9
30	'i'	9
31	'@'	2
32	'2'	
33	';'	
34		
35		
36	'r'	
37	'i'	10
38	'i'	9
39	'@'	2
40	'2'	
41	';'	
42		
43		
44	'}'	
45	'f'	
46	't'	
47	'm'	11
48	'('	

```
49                  ')'
50                  '{'
51                  'd'
52                  't'
53                  'i'                 12
54                  '='
55                  'l'                 13
56                  '@'                  0
57                  '1'
58                  ';'
59
60                  'd'
61                  't'
62                  'i'                 14
63                  ';'
64                  'i'                 14
65                  '='
66                  'i'                 12
67                  'l'                 15
68                  '@'                  5
69                  '2'
70                  ';'
71
72
73                  'I'
74                  'i'                 14
75                  '{'
76                  'p'
77                  'i'                 14
78                  ';'
79                  '}'
80                  'E'
81                  '{'
82                  'p'
83                  'l'                 16
84                  ';'
85                  '}'
86                  'd'
87                  't'
88                  'i'                 17
89                  '='
90                  'l'                 18
91                  '@'                  8
92                  '1'
93                  ';'
94
95                  'I'
96                  'i'                 17
97                  '{'
98                  'p'
99                  'i'                 17
```

100	';'	
101	'}'	
102	'E'	
103	'{'	
104	'p'	
105	'l'	19
106	';'	
107	'}'	
108	'r'	
109	'l'	20
110	';'	
111	'}'	

Таблица идентификаторов

index	name	scope	type	value	id type	lexTable index
0	parseInt		i32	0	function	-1
1	s	parseInt	str	0	parameter	-1
2	concat		str	0	function	-1
3	a	concat	str	0	parameter	-1
4	b	concat	str	0	parameter	-1
5	findSum		i32	0	function	2
6	x	findSum	i32	0	parameter	5
7	y	findSum	i32	0	parameter	8
8	msg		str	0	function	19
9	s	msg	str	0	parameter	22
10	message	msg	str	0	variable	27
11	main		i32	0	function	47
12	x	main	i32	0	variable	53
13	L0		str	'2'	literal	57
14	sum	main	i32	0	variable	62
15	L1		i32	3	literal	70
16	L2		str	'ложь'	literal	83
17	message	main	str	0	variable	88
18	L3		str	'hello '	literal	92
19	L4		str	'empty'	literal	105
20	L5		i32	0	literal	109

Приложение Б

Структуры данных синтаксического анализатора:

```
typedef short GRBALPHABET;

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

namespace GRB {
	struct Rule {
		GRBALPHABET nn;
		int iderror;
		short size;

		struct Chain {
			short size;
			GRBALPHABET* nt;

			Chain();
			Chain(short size, GRBALPHABET s, ...);

			std::string getCChain();
			static GRBALPHABET T(char t) { return GRBALPHABET(t); }
			static GRBALPHABET N(char n) { return -GRBALPHABET(n); }
			static bool isT(GRBALPHABET s) { return s > 0; }
			static bool isN(GRBALPHABET s) { return !isT(s); }
			static char alphabet_to_char(GRBALPHABET s) { return isT(s)
? char(s) : char(-s); }
		} *chains;

		Rule();
		Rule(GRBALPHABET nn, int iderror, short size, Chain c, ...);

		std::string getCRule(short nchain);
		short getNextChain(GRBALPHABET t, Rule::Chain& chain, short n);
	};

	struct Greibach {
		short size;
		GRBALPHABET startN;
		GRBALPHABET stbottomT;
		Rule* rules;

		Greibach();
		Greibach(GRBALPHABET startN, GRBALPHABET stbottomT, short size,
Rule r, ...);
		short getRule(GRBALPHABET nn, Rule& rule) const;
		Rule getRule(short n) const;
	};
	const Greibach getGreibach();
}
```

```
namespace MFST {
template<typename T>
struct use_container : T
{
	using T::T;
	using T::c;
};

typedef use_container<std::stack<short>> MFSTSTACK;

namespace TTM
{
	struct MfstState
	{
		short m_tape_position;
		short m_nrule;
		short m_nrulechain;
		MFSTSTACK m_stack;

		MfstState();
		MfstState(short position, MFSTSTACK m_stack, short m_nrulechain);
		MfstState(short position, MFSTSTACK m_stack, short m_nrule, short
m_nrulechain);
	};

	class SyntaxAnalyzer
	{
	public:
		SyntaxAnalyzer();
		SyntaxAnalyzer(const TTM::LexTable& lextable, const GRB::Grei-
bach& greibach);

		bool Start(Logger& log);
		std::string dumpTrace() const;
		std::string getRules();

	private:
		enum class RC_STEP
		{
			NS_OK,
			NS_NORULE,
			NS_NORULECHAIN,
			NS_ERROR,
			TS_OK,
			TS_NOK,
			TAPE_END,
			SURPRISE
		};

		struct MfstDiagnosis
		{
```

```
                short m_tape_position;
                RC_STEP rc_step;
                short m_nrule;
                short nrule_chain;

                MfstDiagnosis();
                MfstDiagnosis(short m_tape_position, RC_STEP rc_step, short
m_nrule, short nrule_chain);
           } diagnosis[MFST_DIAGN_NUMBER];

           GRBALPHABET* m_tape;
           short m_tape_position;
           short m_nrule;
           short m_nrulechain;
           short m_tape_size;
           GRB::Greibach greibach;
           const TTM::LexTable& lextable;
           MFSTSTACK m_stack;
           use_container<std::stack<MfstState>> m_storestate;
           std::stringstream m_trace;
           std::stringstream m_rules;

           std::string getCSt();
           std::string getCTape(short pos, short n = 25);
           bool save_state();
           bool restore_state();
           bool push_chain(GRB::Rule::Chain chain);
           RC_STEP step();
           bool savediagnosis(RC_STEP rc_step);
     };
}
```

Грамматика Грейбах:

```
const Greibach greibach(NS('S'), TS('$'),
                6,
                Rule(NS('S'), GRB_ERROR_SERIES + 0,
                     2,
                     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(11, TS('f'), TS('t'), TS('m'), TS('('),
TS(')'), TS('{'), NS('N'), TS('r'), NS('E'), TS(';'), TS('}'))
                ),
                Rule(NS('N'), GRB_ERROR_SERIES + 1,
                     8,
                     Rule::Chain(),
                     Rule::Chain(5, TS('d'), TS('t'), TS('i'), TS(';'),
NS('N')),
                     Rule::Chain(5, TS('i'), TS('='), NS('E'), TS(';'),
NS('N')),
                     Rule::Chain(7, TS('d'), TS('t'), TS('i'), TS('='),
NS('E'), TS(';'), NS('N')),

                     Rule::Chain(4, TS('p'), TS('i'), TS(';'), NS('N')),
                     Rule::Chain(4, TS('p'), TS('l'), TS(';'), NS('N')),

                     Rule::Chain(6, TS('I'), NS('E'), TS('{'), NS('N'),
TS('}'), NS('N')),
                     Rule::Chain(10, TS('I'), NS('E'), TS('{'), NS('N'),
TS('}'), TS('E'), TS('{'), NS('N'), TS('}'), NS('N'))
                ),
                Rule(NS('E'), GRB_ERROR_SERIES + 2,
                     9,
                     Rule::Chain(1, TS('i')),
                     Rule::Chain(1, TS('l')),
                     Rule::Chain(3, TS('('), NS('E'), TS(')')),
                     Rule::Chain(4, TS('i'), TS('('), NS('W'), TS(')')),
                     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(5, TS('i'), TS('('), NS('W'), TS(')'),
NS('M')),
                     Rule::Chain(4, TS('i'), TS('('), NS('W'), TS(')'))
                ),
                Rule(NS('M'), GRB_ERROR_SERIES + 3,
                     20,
                     Rule::Chain(2, TS('+'), NS('E')),
                     Rule::Chain(4, TS('+'), TS('('), NS('E'), TS(')')),
                     Rule::Chain(5, TS('+'), TS('('), NS('E'), TS(')'),
NS('M')),
                     Rule::Chain(3, TS('+'), NS('E'), NS('M')),

                     Rule::Chain(2, TS('-'), NS('E')),
```

```
                    Rule::Chain(4, TS('-'), TS('('), NS('E'), TS(')')),
                    Rule::Chain(5, TS('-'), TS('('), NS('E'), TS(')'),
NS('M')),
                    Rule::Chain(3, TS('-'), NS('E'), NS('M')),

                    Rule::Chain(2, TS('*'), NS('E')),
                    Rule::Chain(4, TS('*'), TS('('), NS('E'), TS(')')),
                    Rule::Chain(5, TS('*'), TS('('), NS('E'), TS(')'),
NS('M')),
                    Rule::Chain(3, TS('*'), NS('E'), NS('M')),

                    Rule::Chain(2, TS('/'), NS('E')),
                    Rule::Chain(4, TS('/'), TS('('), NS('E'), TS(')')),
                    Rule::Chain(5, TS('/'), TS('('), NS('E'), TS(')'),
NS('M')),
                    Rule::Chain(3, TS('/'), NS('E'), NS('M')),

                    Rule::Chain(2, TS('%'), NS('E')),
                    Rule::Chain(4, TS('%'), TS('('), NS('E'), TS(')')),
                    Rule::Chain(5, TS('%'), TS('('), NS('E'), TS(')'),
NS('M')),
                    Rule::Chain(3, TS('%'), NS('E'), NS('M'))
               ),
               Rule(NS('F'), GRB_ERROR_SERIES + 4,
                    2,
                    Rule::Chain(2, TS('t'), TS('i')),
                    Rule::Chain(4, TS('t'), TS('i'), TS(','), NS('F'))
               ),
               Rule(NS('W'), GRB_ERROR_SERIES + 5,
                    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'))
               )
          );
```

Дерево разбора

```
Шаг : Правило                    Входная лента                Стек
0   : S->fti(F){NrE;}S           fti(ti,ti){ri+i;}fti(ti){    S$
1   : SAVESTATE:          1
1   :                            fti(ti,ti){ri+i;}fti(ti){    fti(F){NrE;}S$
2   :                            ti(ti,ti){ri+i;}fti(ti){d    ti(F){NrE;}S$
3   :                            i(ti,ti){ri+i;}fti(ti){dt    i(F){NrE;}S$
4   :                            (ti,ti){ri+i;}fti(ti){dti    (F){NrE;}S$
5   :                            ti,ti){ri+i;}fti(ti){dti=    F){NrE;}S$
6   : F->ti                      ti,ti){ri+i;}fti(ti){dti=    F){NrE;}S$
7   : SAVESTATE:          2
7   :                            ti,ti){ri+i;}fti(ti){dti=    ti){NrE;}S$
8   :                            i,ti){ri+i;}fti(ti){dti=i    i){NrE;}S$
9   :                            ,ti){ri+i;}fti(ti){dti=i(    ){NrE;}S$
10  : TS_NOK/NS_NORULECHAIN
10  : RESTORESTATE
10  :                            ti,ti){ri+i;}fti(ti){dti=    F){NrE;}S$
11  : F->ti,F                    ti,ti){ri+i;}fti(ti){dti=    F){NrE;}S$
12  : SAVESTATE:          2
12  :                            ti,ti){ri+i;}fti(ti){dti=    ti,F){NrE;}S$
13  :                            i,ti){ri+i;}fti(ti){dti=i    i,F){NrE;}S$
14  :                            ,ti){ri+i;}fti(ti){dti=i(    ,F){NrE;}S$
15  :                            ti){ri+i;}fti(ti){dti=i(i    F){NrE;}S$
16  : F->ti                      ti){ri+i;}fti(ti){dti=i(i    F){NrE;}S$
17  : SAVESTATE:          3
17  :                            ti){ri+i;}fti(ti){dti=i(i    ti){NrE;}S$
18  :                            i){ri+i;}fti(ti){dti=i(i,    i){NrE;}S$
19  :                            ){ri+i;}fti(ti){dti=i(i,i    ){NrE;}S$
```

Окончание дерева разбора

```
376 :                                pl;}rl;}                    N}NrE;}$
377 : N->pi;N                        pl;}rl;}                    N}NrE;}$
378 : SAVESTATE:          40
378 :                                pl;}rl;}                    pi;N}NrE;}$
379 :                                l;}rl;}                     i;N}NrE;}$
380 : TS_NOK/NS_NORULECHAIN
380 : RESTORESTATE
380 :                                pl;}rl;}                    N}NrE;}$
381 : N->pl;N                        pl;}rl;}                    N}NrE;}$
382 : SAVESTATE:          40
382 :                                pl;}rl;}                    pl;N}NrE;}$
383 :                                l;}rl;}                     l;N}NrE;}$
384 :                                ;}rl;}                      ;N}NrE;}$
385 :                                }rl;}                       N}NrE;}$
386 : N->                            }rl;}                       N}NrE;}$
387 : SAVESTATE:          41
387 :                                }rl;}                       }NrE;}$
388 :                                rl;}                        NrE;}$
389 : N->                            rl;}                        NrE;}$
390 : SAVESTATE:          42
390 :                                rl;}                        rE;}$
391 :                                l;}                         E;}$
392 : E->l                           l;}                         E;}$
393 : SAVESTATE:          43
393 :                                l;}                         l;}$
394 :                                ;}                          ;}$
395 :                                }                           }$
396 :                                                            $
397 : TAPE_END
398 : ------>TAPE_END
```

Приложение В
Ошибки семантического анализатора:

Пример 1

```
fn i32 main()
{
let i32 num = parseInt(3,4);
ret 0;
};
```

Ошибка 701: Ошибка в параметрах функции

Пример 2

```
main
{
let i32 num = ‘num’;
echo num;
ret 0;
};
```

Ошибка 706: Несоответсвие присваимаего типа данных

Пример 3

```
main
{
ret '0';
};
```

Ошибка 700: Ошибка в возвращаемом значении

Приложение Г

Программная реализация обработки выражений:

```
TTM::PolishNotation::PolishNotation(LexTable& lextable, IdTable& idtable)
	:lextable(lextable), idtable(idtable)
{
	for (int i = 0; i < lextable.size(); ++i)
	{
		if (lextable[i].lexeme == LEX_ASSIGN || lextable[i].lexeme == 
LEX_RET)
		{
			convert(i + 1);
		}
	}
}

int TTM::PolishNotation::getOperationsPriority(char operation)
{
	if (operation == LEX_OPENING_PARENTHESIS || operation == LEX_CLOS-
ING_PARENTHESIS)
		return 1;
	else if (operation == LEX_PLUS || operation == LEX_MINUS)
		return 2;
	else if (operation == LEX_ASTERISK || operation == LEX_SLASH || opera-
tion == LEX_PERCENT)
		return 3;
	return EOF;
}

bool TTM::PolishNotation::convert(int startIndex)
{
	std::vector<LexTable::Entry> infixExpressionEntries;
	int operandsCounter = 0, operationsCounter = 0;
	bool functionParameters = false;

	for (int i = startIndex; i < lextable.size() && lextable[i - 1].lexeme 
!= LEX_SEMICOLON; ++i)
	{
		char& lexeme = lextable[i].lexeme;
		if (lextable[i].idTableIndex != TI_NULLIDX && 
idtable[lextable[i].idTableIndex].idType == it::id_type::function)
		{
			lexeme = LEX_FUNCTION_CALL;
			functionParameters = true;
			++operandsCounter;
		}
		else if (lexeme == LEX_CLOSING_PARENTHESIS && functionParameters)
		{
			functionParameters = false;
		}
```

```
            else if (lexeme == LEX_PLUS || lexeme == LEX_MINUS || lexeme ==
LEX_ASTERISK || lexeme == LEX_SLASH || lexeme == LEX_PERCENT)
            {
                  ++operationsCounter;
            }
            else if (!functionParameters && (lexeme == LEX_ID || lexeme ==
LEX_LITERAL))
            {
                  ++operandsCounter;
            }

            infixExpressionEntries.push_back(lextable[i]);
      }

      if (operandsCounter - operationsCounter != 1)
            return false;

      std::vector<LexTable::Entry> postfixExpressionEntries = getPostfixEx-
pression(infixExpressionEntries);
      for (size_t i = 0; i < infixExpressionEntries.size(); ++i) {
            if (i < postfixExpressionEntries.size()) {
                  lextable[i + startIndex] = postfixExpressionEntries[i];
            }
            else {
                  lextable[i + startIndex] = { FORBIDDEN_SYMBOL, EOF, EOF };
            }
      }

      return true;
}

std::vector<TTM::LexTable::Entry> TTM::PolishNotation::getPostfixExpres-
sion(const std::vector<LexTable::Entry>& entries)
{
      std::vector<LexTable::Entry> output;
      output.reserve(entries.size());
      std::stack<LexTable::Entry> stack;
      bool foundFunction = false;

      for (const auto& e : entries)
      {
            if (e.lexeme == LEX_PLUS || e.lexeme == LEX_MINUS || e.lexeme ==
LEX_ASTERISK
                  || e.lexeme == LEX_SLASH || e.lexeme == LEX_PERCENT) {
                  if (!stack.empty() && stack.top().lexeme != LEX_OPENING_PA-
RENTHESIS) {
                        while (!stack.empty() && getOperationsPriority(e.lex-
eme) <= getOperationsPriority(stack.top().lexeme))
                        {
                              output.push_back(stack.top());
                              stack.pop();
```

```
                    }
                }

                stack.push(e);
            }
            else if (e.lexeme == LEX_COMMA)
            {
                while (!stack.empty() && stack.top().lexeme != LEX_OPEN-
ING_PARENTHESIS)
                {
                    output.push_back(stack.top());
                    stack.pop();
                }
            }
            else if (e.lexeme == LEX_FUNCTION_CALL)
            {
                foundFunction = true;
                stack.push(e);
            }
            else if (e.lexeme == LEX_OPENING_PARENTHESIS)
            {
                stack.push(e);
            }
            else if (e.lexeme == LEX_CLOSING_PARENTHESIS)
            {
                while (stack.top().lexeme != LEX_OPENING_PARENTHESIS)
                {
                    output.push_back(stack.top());
                    stack.pop();
                }
                stack.pop();

                if (!stack.empty() && stack.top().lexeme == LEX_FUNC-
TION_CALL)
                {
                    output.push_back(stack.top());
                    output.push_back({ getFunctionParametersCountBy-
Name(idtable[stack.top().idTableIndex].name), TI_NULLIDX, TI_NULLIDX });
                    stack.pop();
                    foundFunction = false;
                }
            }
            else if (e.lexeme != LEX_SEMICOLON)
            {
                output.push_back(e);
            }
        }

        while (!stack.empty())
        {
            output.push_back(stack.top());
```

```
            stack.pop();
      }

      output.push_back(entries.back());

      return output;
}

char TTM::PolishNotation::getFunctionParametersCountByName(const
std::string& functionName)
{
      int parametersCount = 0;
      int index = idtable.getIdIndexByName("", functionName);
      for (int i = index + 1; idtable[i].idType == it::id_type::parameter;
++i)
      {
            ++parametersCount;
      }

      return parametersCount + '0';
}
```

Приложение Д
Результат генерации кода:

```
.586
.model flat, stdcall
includelib libucrt.lib
includelib kernel32.lib
ExitProcess PROTO : SDWORD
includelib ../Debug/stdlib.lib
_echoInt PROTO : SDWORD
_echoStr PROTO : SDWORD
_parseInt PROTO : SDWORD
_concat PROTO : SDWORD, : SDWORD

.stack 4096

.const
_DIVIDE_BY_ZERO_EXCEPTION BYTE 'деление на 0',0
_L0 BYTE '2', 0
_L1 SDWORD 3
_L2 BYTE 'ложь', 0
_L3 BYTE 'hello ', 0
_L4 BYTE 'empty', 0
_L5 SDWORD 0

.data
_msgmessage SDWORD 0
_mainx SDWORD 0
_mainsum SDWORD 0
_mainmessage SDWORD 0

.code
_findSum PROC _findSumx : SDWORD, _findSumy : SDWORD
push _findSumx
push _findSumy
pop eax
pop ebx
add eax, ebx
push eax
pop eax
ret 8
_findSum ENDP

_msg PROC _msgs : SDWORD
push _msgs
invoke _concat, _msgs, _msgs
push eax
pop _msgmessage
push _msgs
invoke _concat, _msgmessage, _msgs
push eax
```

```
pop eax
ret 4
_msg ENDP

_main PROC
invoke _parseInt, offset _L0
push eax
pop _mainx
push _L1
invoke _findSum, _mainx, _L1
push eax
pop _mainsum
.if _mainsum != 0
push _mainsum
call _echoInt
.else
push offset _L2
call _echoStr
.endif
invoke _msg, offset _L3
push eax
pop _mainmessage
.if _mainmessage != 0
push _mainmessage
call _echoStr
.else
push offset _L4
call _echoStr
.endif
push _L5
call ExitProcess
_main ENDP

end _main
```

Заключение

В данном курсовом проекте были выполнены поставленные минимальные требования. В ходе работы было изучено много нового, а также закреплены знания, которые были получены ранее. Также стоит отметить, что данный курсовой проект позволил совместить закрепление знаний сразу по двум языкам программирования, таких как C++ и Ассемблер. При написании приложения были усвоены такие понятия как синтаксический, лексический и семантический анализатор и т.п. В итоге был получен примитивный язык программирования TTM-2020, который не имеет сложных конструкций, которые реализованы на сегодняшний день во многих других языках программирования.

Окончательная версия языка TTM-2020 включает:

1) 2 типа данных;
2) Поддержка оператора вывода;
3) Возможность подключения и вызова функций стандартной библиотеки;
4) Наличие 5 арифметических операторов для вычисления выражений;
5) Возможность вызова функции в выражении

Литература

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

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

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

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

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

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

7. Смелов, В. В. Введение в объектно-ориентированное программирование на C++ / В. В. Смелов.– Минск : БГТУ, 2009. - 94с.

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