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 BVD-2021»

Выполнил студент Буранко Валерия Дмитриевна
(Ф.И.О.)

Руководитель проекта асс. Пахолко Алёна 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.)

Консультант асс. Пахолко Алёна Степановна
(учен. степень, звание, должность, подпись, Ф.И.О.)

Нормоконтролер асс. Пахолко Алёна Степановна
(учен. степень, звание, должность, подпись, Ф.И.О.)

Курсовой проект защищен с оценкой

Минск 2021

Содержание

Введение

Целью курсового проекта является разработка транслятора для своего языка программирования: BVD-2021.

Транслятор — это комплекс отдельных программ, позволяющих преобразовывать исходный код на одном языке программирования в исходный код на другом языке программирования. Задача транслятора — сделать программу, написанную на некотором языке программирования, понятной компьютеру. Этого можно добиться одним из двух способов: компиляцией или интерпретацией. Язык BVD-2021 является компилируемым. Компилятор переводит исходную программу в эквивалентную ей на язык ассемблера.

Процесс создания компилятора можно свести к решению нескольких задач, которые принято называть фазами компиляции. Разрабатываемый компилятор состоит из следующих фаз:

- лексический анализ;
- синтаксический анализ;
- семантический анализ;
- генерация кода.

Исходя из цели курсового проекта, были определены следующие основные задачи:

- разработка спецификации языка программирования;
- разработка 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 BVD-2021 классифицируется как процедурный, универсальный, строготипизированный, компилируемый и не объектно-ориентированный язык.

1.2 Алфавит языка

Алфавит языка BVD-2021 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 двоичной системы счисления 1 и 0, спецсимволы, а также непечатные символы пробела, табуляции и перевода строки. Русские символы разрешены только в строковых литералах.

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
(…)	Круглые скобки	Приоритет операций, параметры функции, условие
“…”	Двойные кавычки	Строковые литералы
=	Знак «равно»	Присваивание значения
,	Запятая	Разделение параметров
+ - * / %	Знаки «плюс», «минус», «умножение», «деление», «деление по модулю»,	Выражения
& ! > <	Знаки «равно», «не равно», «больше», «меньше»	Условия

1.4 Применяемые кодировки

Для написания исходного кода на языке программирования BVD-2021 используется кодировка Windows-1251.

1.5 Типы данных

В языке BVD-2021 реализованы два типа данных: целочисленный и строковый. Описание реализованных типов данных представлено в таблице 1.2.

Таблица 1.2 — Типы данных языка BVD-2021

Тип данных	Описание
byte (целый)	Знаковый целый тип byte имеет размер 1 байт. Минимальное значение -128, максимальное значение 127. Инициализация по умолчанию: значение 0.
str (строковый)	Указатель, занимает в памяти: 4 байт. Инициализация по умолчанию: 0.

1.6 Преобразование типов данных

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

1.7 Идентификаторы

В имени идентификатора допускаются только символы латинского алфавита нижнего регистра. Максимальная длина имени идентификатора – 10 символов. При вводе идентификатора длиной более разрешенного количества символов, он будет усекаться. Имя идентификатора не может совпадать с ключевыми словами и не может иметь имя, которое совпадает с именем функции стандартной библиотеки.

1.8 Литералы

С помощью литералов осуществляется инициализация переменных. В языке существует два типа литералов. Краткое описание литералов языка BVD-2021 представлено в таблице 1.3.

Таблица 1.3 — Описание литералов

Тип литерала	Регулярное выражение	Описание	Пример
Целочисленный литерал	$[0\text{-}1]^{+}b$	Целочисленные литералы в двоичном представлении. Литералы могут быть только rvalue.	declare byte sum; sum = 101b; 101b – целочисленный литерал.
Строковые литерал	$[a\text{-}z\|A\text{-}Z\|A\text{-}Я\|a\text{-}я\|0\text{-}9\|!\text{-}/]^{+}$	Символы, заключённые в “…” (двойные кавычки). Литералы могут быть только rvalue.	declare str text; text = “text”; “text” – строковый литерал.

1.9 Область видимости идентификаторов

Область видимости «сверху вниз» (по принципу C++). В языке BVD-2021 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 номер идентификатора функции в которой она объявлена.

1.10 Инициализация данных

При объявлении переменной не допускается инициализация данных. Краткое описание способов инициализации переменных языка BVD-2021 представлено в таблице 1.4.

Таблица 1.4 — Способы инициализации переменных

Конструкция	Описание	Пример
declare <тип данных> <идентификатор>;	Автоматическая инициализация: переменные типа byte инициализируются нулём, переменные типа str – указателем на строку.	declare byte sum; declare str text;
<идентификатор> = <значение>;	Присваивание переменной значения.	sum = 7; string = “text”;

1.11 Инструкции языка

Все возможные инструкции языка программирования BVD-2021 представлены в общем виде в таблице 1.5.

Таблица 1.5 — Инструкции языка программирования BVD-2021

Инструкция	Запись на языке BVD-2021
Объявление переменной	declare <тип данных> <идентификатор>;
Присваивание	<идентификатор> = <значение>/<идентификатор>/<выражение>;
Объявление внешней функции	declare <тип данных> function <идентификатор> (<тип данных> <идентификатор>, …)

Блок инструкций	{
	…
	}

Возврат из подпрограммы	return <идентификатор> / <литерал> / <выражение>;
Условная инструкция	if (условие) {/программный блок если условие верно/} else {/программный блок если условие ложно/}
Цикл	while(условие) {/программный блок если условие верно/}/операторы если условие ложно/
Вывод целочисленных данных	output <идентификатор> / <литерал> / <выражение>;
Вывод строковых данных	output <идентификатор> / <литерал> / <выражение>;

1.12 Операции языка

Язык программирования BVD-2021 может выполнять операции, представленные в таблице 1.6.

Таблица 1.6 — Операции языка программирования BVD-2021

Операция	Примечание	Типы данных	Пример
=	Присваивание	(byte, byte) (str, str)	sum = 1010b; text =”text”;
( )	Приоритет операций	-	sum = (a + b) * c;
+	Сложение	(byte, byte)	a = b + c;
-	Вычитание	(byte, byte)	a = b - c;
*	Умножение	(byte, byte)	a = b * c;
/	Деление	(byte, byte)	a = b / c;
%	Остаток от деления	(byte, byte)	a = b % c;

1.13 Выражения и их вычисления

Круглые скобки в выражении используются для изменения приоритета операций. Выражение может содержать вызов функции.

1.14 Программные конструкции языка

Ключевые программные конструкции языка программирования BVD-2021 представлены в таблице 1.7. Для цикла и условной конструкции нельзя сравнивать два литерала, строки.

Таблица 1.7 — Программные конструкции языка BVD-2021

Конструкция	Запись на языке BVD-2021

Главная функция (точка входа)	major
	{
	…
	return <идентификатор> / <литерал> / <выражение>;
	}

Функции	<тип данных> function <идентификатор>(<тип данных> <идентификатор>, …)
	{
	…
	return <идентификатор> / <литерал> / <выражение>;
	};

Цикл	while (<идентификатор> / <литерал> <условный знак> <идентификатор> / <литерал>)
	{
	…
	}

Условная конструкция	if (<идентификатор> / <литерал> <условный знак> <идентификатор> / <литерал>)
	{ … }
	else
	{ … }

1.15 Область видимости

В языке BVD-2021 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	Ошибки
1	Повторное объявление идентификатора
2	Формальные и фактические параметры функции не совпадают
3	Типы идентификаторов выражения не совпадают
4	Функция должна иметь параметры
5	К строкам нельзя применять операции
6	Типы сравниваемых значений должны быть byte
7	Объявлять внутри условных конструкциях и циклах запрещено
8	Переменная должна быть определена перед её использованием
9	Последний оператор функции должен быть return
10	Определенная функция не соответствует функциям стандартной библиотеки
11	Главная функция должна быть задана один раз
12	Максимальное число параметров функции 5

1.17 Распределение оперативной памяти на этапе выполнения

Все переменные размещаются в стеке.

1.18 Стандартная библиотека и её состав

Стандартная библиотека BVD-2021 написана на языке программирования MASM Функции стандартной библиотеки с описанием представлены в таблице 1.9.

Таблица 1.9 — Состав стандартной библиотеки

Функция (MASM)	Возвращаемое значение	Описание
`_abs PROC a: BYTE _`	byte	Функция вычисления модуля числа
`_pow PROC a: BYTE, b: BYTE`	byte	Функция возведения числа a в степень b
`_strcat PROC str1: DWORD, str2: DWORD`	dword	Конкатенация двух строк

1.19 Ввод и вывод данных

В языке BVD-2021 не реализованы средства ввода данных.

Для вывода данных в стандартный поток вывода предусмотрен оператор output.

1.20 Точка входа

В языке BVD-2021 каждая программа должна содержать главную функцию major, т. е. точку входа, с которой начнется последовательное выполнение программы.

1.21 Препроцессор

Препроцессор в языке программирования BVD-2021 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

BVD-2021 транслируется в язык ассемблера.

1.24 Классификация сообщений транслятора

В случае возникновения ошибки в коде программы на языке BVD-2021 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
120-125	Ошибки лексического анализа
600-609	Ошибки синтаксического анализа
610-622	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.

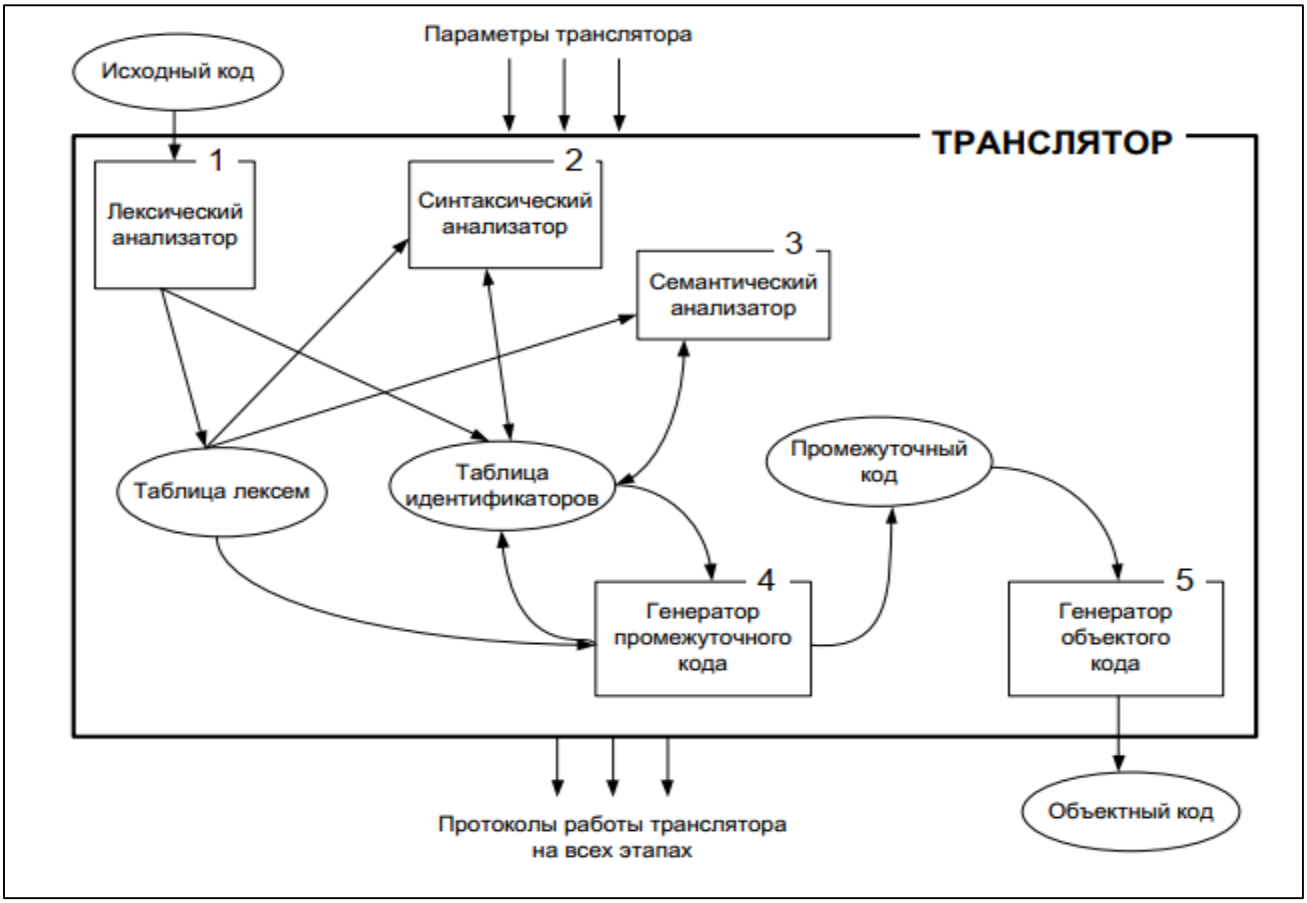

Рисунок 2.1 — Структура транслятора

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 исходный код на языке BVD-2021. 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 он выявляет синтаксические ошибки, допущенные в исходном коде. Формирует дерево разбора, а также выводит трассировку (разбор) цепочек.

Семантический анализатор — отвечает за проверку соблюдения семантических правил языка. Состоит из двух частей: первая часть включена в лексический

анализ, при заполнении таблиц; вторая является отдельной функцией, выполняемая после синтаксической проверки, использующая таблицы идентификаторов и лексем, а также дерево разбора.

Генератор кода — принимает на вход таблицу идентификаторов и таблицу лексем. Задача этого компонента заключается в трансляции, уже пройденного все предыдущие этапы кода на языке BVD-2021, 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 BVD-2021

Входной параметр	Описание	Значение по умолчанию
-in:<имя_файла>	Входной файл с расширением .txt, в котором содержится исходный код языка BVD-2021	Не предусмотрено
-log:<имя_файла>	Файл, содержащий информацию о работе транслятора.	<имя_файла>.log
-out:<имя_файла>	Файл для записи результата работы лексического и синтаксического анализаторов.	<имя_файла>.out

2.3 Перечень протоколов, формируемых транслятором и их содержимое

Протоколы транслятора BVD-2021 представлены в таблице 2.2.

Таблица 2.2 — Протоколы транслятора BVD-2021

Протокол	Содержимое
<имя_файла>.log	Содержит служебную информацию, получаемую в ходе работы транслятора, а также ошибки, которые возникают на этапе обработки исходного кода.
<имя_файла>.out	Содержит текстовую информацию таблицы лексем и таблицы идентификаторов.

Протокол работы нужен для отображения хода выполнения трансляции языка BVD-2021. 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. Каждому идентификатору и литералу в таблице лексем сопоставляется ее тип и запись в таблице идентификаторов, в которой хранится дополнительная информация. Структура лексического анализатора представлена на рисунке 3.1.

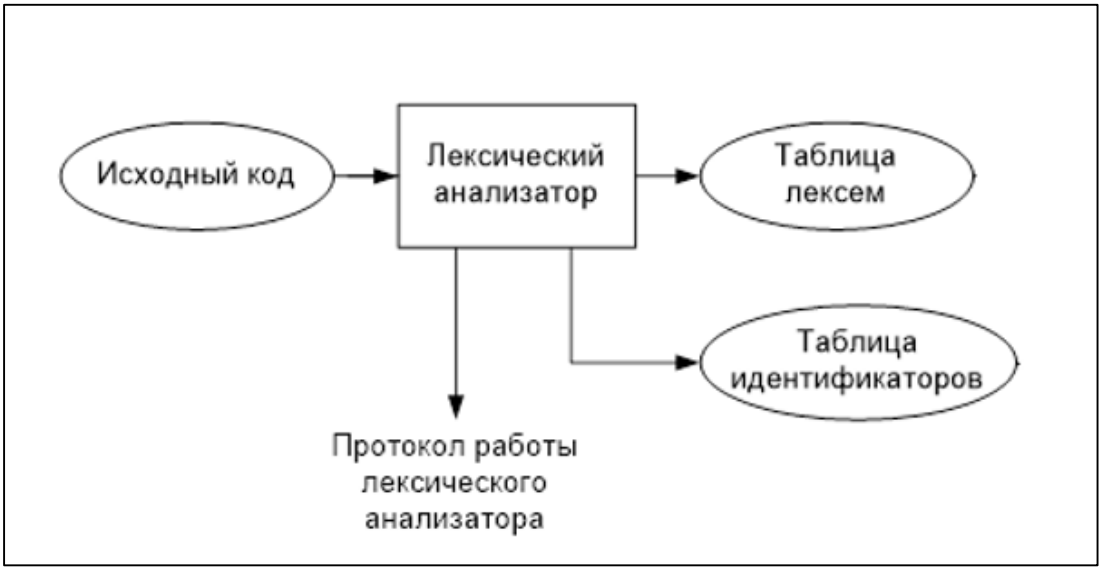

Рисунок 3.1 — Структура лексического анализатора BVD-2021

Результатом работы лексического анализатора являются таблица лексем и идентификаторов.

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::T, IN::T, IN::T, IN::T, IN::T, IN::T, IN::T, IN::T, IN::T, IN::T, IN::T, IN::T, IN::T, IN::T, IN::T, IN::T,\
IN::T, IN::T, IN::T, IN::T, IN::T, IN::T, IN::T, IN::T, IN::T, IN::T, IN::T, IN::T, IN::T, IN::T, IN::T, IN::T,\
IN::T, IN::T, IN::T, IN::T, IN::T, IN::T, IN::T, IN::T, IN::T, IN::T, IN::T, IN::T, IN::T, IN::T, IN::T, IN::T,\
IN::T, IN::T, IN::T, IN::T, IN::T, IN::T, IN::T, IN::T, IN::T, IN::T, IN::T, IN::T, IN::T, IN::T, IN::T, IN::T,\
IN::T, IN::T, IN::T, IN::T, IN::T, IN::T, IN::T, IN::T, IN::T, IN::T, IN::T, IN::T, IN::T, IN::T, IN::T, IN::T,\
IN::T, IN::T, IN::T, IN::T, IN::T, IN::T, IN::T, IN::T, IN::I, IN::T, IN::T, IN::T, IN::T, IN::T, IN::T, IN::T,\
IN::T, IN::T, IN::T, IN::T, IN::T, IN::T, IN::T, IN::T, IN::T, IN::T, IN::T, IN::T, IN::T, IN::T, IN::T, IN::T,\
IN::T, IN::T, IN::T, IN::T, IN::T, IN::T, IN::T, IN::T, IN::T, IN::T, IN::T, IN::T, IN::T, IN::T, IN::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, 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,\
}
// 0    1      2      3     4      5      6      7    8      9     A      B     C      D     E      F
```

Рисунок 3.2. — Таблица контроля входных символов

Описание значения символов: T — разрешённый символ, F — запрещённый символ, I — игнорируемый символ.

3.3 Удаление избыточных символов

Избыточными символами являются идущие подряд пробелы. Игнорируются на этапе записи текста из файла.

На этапе разбиения исходного кода на лексемы избыточными символами являются пробелы, символы конца строки и символы табуляции, в таблицу лексем не добавляются, однако влияют на запись информации.

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.

Таблица 3.1 — Таблица соответствий цепочек, их типов и лексем

Цепочка	Тип	Лексема
byte	Ключевое слово	t
str	Ключевое слово	t
Идентификатор	Переменная, функция или параметр	i
Целочисленный литерал	Данные	l
Строковый литерал	Данные	l

Окончание таблицы 3.1

major	Ключевое слово	m
function	Ключевое слово	f
declare	Ключевое слово	d
return	Ключевое слово	r
output	Ключевое слово	p
while	Ключевое слово	w
if	Ключевое слово	1
else	Ключевое слово	2
+ - * / % =	Операторы	+ - * / % =
> < & !	Операторы сравнения	> < & !
; , {} ()	Сепараторы	; , {} ()

Пример реализации таблицы лексем представлен в приложении А.

3.5 Основные структуры данных

Основные структуры таблиц лексем и идентификаторов данных языка BVD-2021, используемых для хранения, представлены в приложении А. В таблице лексем содержится лексема, её номер, полученный при разборе, номер строки в исходном коде, индекс в таблице идентификаторов, тип данных. В таблице идентификаторов содержится имя идентификатора, индекс первой строки в таблице лексем, индекс функции, которой принадлежит идентификатор, тип данных, тип идентификатора, типы параметров, значение, определяющее было ли присвоено какое либо значение идентификатору.

3.6 Принцип обработки ошибок

В случае если в результате работы лексического анализатора найдена ошибка, анализатор добавляет информацию в таблицу ошибок.

После чего главная функция записывает ошибки в файл протокола. Они включают следующую информацию: код ошибки, номер строки в коде, номер позиции в строке или только код ошибки. Пользователь может ознакомиться со всеми ошибками, полученными в результате анализа, открыв протокол. Если есть ошибки, выполнение программы завершается.

3.7 Структура и перечень сообщений лексического анализатора

Возможные ошибки лексического анализатора представлено на рисунке 3.3.

```
ERROR_ENTRY(120, "Ошибка при распознавании лексемы"),
ERROR_ENTRY(121, "Идентификатор не определен"),
ERROR_ENTRY(122, "Строка превышает максимально допустимый размер"),
ERROR_ENTRY(123, "Таблица лексем переполнена"),
ERROR_ENTRY(124, "Таблица идентификаторов переполнена"),
ERROR_ENTRY(125, "Идентификатор уже занят в стандартной библиотеке"),
```

Рисунок 3.3 — Ошибки лексического анализатора

3.8 Параметры лексического анализатора и режимы его работы

Входным параметром лексического анализа являются редактированный код исходного файла, пустая таблица лексем, пустая таблица идентификаторов, таблица ошибок.

Лексический анализатор обрабатывает код исходного файла, проверяя символы и слова на соответствие возможным цепочкам. Заполняет таблицу лексем и таблицу идентификаторов.

3.9 Алгоритм лексического анализа

Лексический анализатор проверяет входной поток символов программы на соответствие возможным цепочкам, выполняя функцию распознавания лексем, используя конечный автомат. При успешном распознавании информация о выделенной лексеме заносится в таблицу лексем и таблицу идентификаторов (если подходящий тип), алгоритм возвращается к началу. При неуспешном распознавании дополняется таблица ошибок, не распознанная лексема игнорируется, делается попытка распознать следующую лексему.

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.

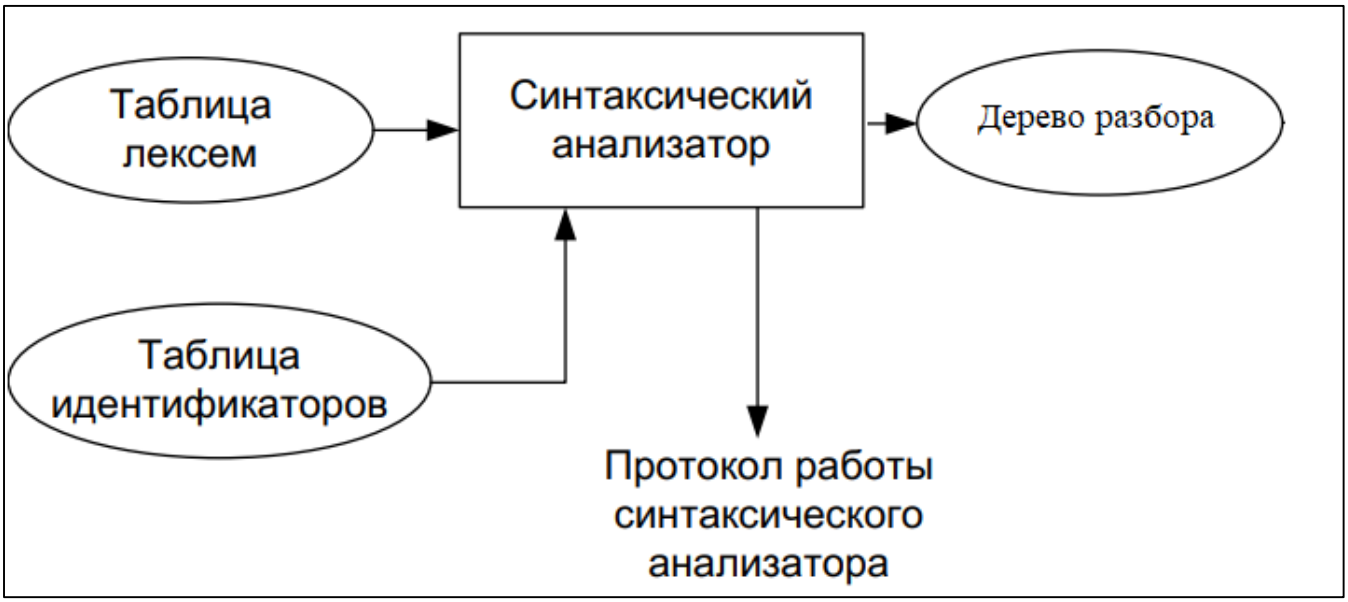

Рисунок 4.1 — Структура синтаксического анализатора

4.2 Контекстно-свободная грамматика, описывающая синтаксис языка

В синтаксическом анализаторе транслятора языка BVD-2021 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 BVD-2021 представлена в приложении Б.

TS — терминальные символы, NS — нетерминальные символы.

Таблица 4.1 — Перечень правил, составляющих грамматику языка и описание нетерминальных символов BVD-2021

Нетерминал	Цепочки правил	Описание
S	m{N}; m{N};S tfi(F){N}; tfi(F){N};S	Правила для общих структур

N	dti;
	rE;
	i=E;
	dtfi(F);
	dti;N
	rE;N
	i=E;N
	dtfi(F);N
	c(W);
	c(W);N
	w(C){N}
	w(C){N}N
	1(C){N}
	1(C){N}N
	1(C){N}2{N}
	1(C){N}2{N}N
		Правила для конструкций и инструкций

E	i l (E) i(W) iM lM (E)M i(W)M	Правила для выражений
M	+E -E *E /E %E +EM -EM *EM /EM %EM	Правила для операторов
F	ti ti,F	Правила для параметров функции

Окончание таблицы 4.1

W	i l i,W l,W	Правила для параметров вызываемой функции
C	i>i i<i i&i i!i i>l i<l i&l i!l l>i l<i l&i l!i	Правила для условия

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 BVD-2021. 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.

Работа распознавателя:

1) если верхний символ магазина (вершина стека) МП-автомата является нетерминальный символом, то его можно заменить на цепочку терминальных и нетерминальных символов при условии, что в грамматике языка есть соответствующее правила. Считывающая головка автомата при этом не сдвигается (этот шаг работы называется выбор правила);

2) если верхний символ магазина (вершина стека) является терминальным символом, который совпадает с текущим символом входной цепочки, то этот символ выталкивается из стека и считывающая головка передвигается на одну позицию вправо.

.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 параметрах функции"),
ERROR_ENTRY(604, "Ошибка в параметрах вызываемой функции"),
ERROR_ENTRY(605, "Ошибка в условии"),
```

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 BVD-2021 представлен в приложении Б.

Глава 5. Разработка семантического анализатора

5.1. Структура семантического анализатора

Первая часть семантического анализа проходит во время лексического разбора, и проверяет некоторые правила, вторая происходит после выполнения синтаксического анализа и реализуется в виде проверки дерева разбора, таблиц литералов, идентификаторов на соответствие правилам семантики.

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(610, "Повторное объявление идентификатора"),
ERROR_ENTRY(611, "Определенная функция не соответствует функциям стандартной библиотеки"),
ERROR_ENTRY(612, "Формальные и фактические параметры функции не совпадают"),
ERROR_ENTRY(613, "Типы идентификаторов выражения не совпадают"),
ERROR_ENTRY(614, "Функция должна иметь параметры"),
ERROR_ENTRY(615, "К строкам нельзя применять операции"),
ERROR_ENTRY(616, "Типы сравниваемых значений должны быть byte"),
ERROR_ENTRY(617, "Объявлять внутри if else и while блоков запрещено"),
ERROR_ENTRY(618, "Переменная должна быть определена перед её использованием"),
ERROR_ENTRY(619, "Последний оператор функции должен быть return"),
ERROR_ENTRY(620, "Типы параметров должны совпадать с типом функции"),
ERROR_ENTRY(621, "Максимальное число параметров функции 5"),
ERROR_ENTRY(622, "Главная функция major должна быть задана один раз"),
```

Рисунок 5.1 — Перечень сообщений семантического анализатора

5.4 Принцип обработки ошибок

В качестве входных параметров выступают таблица литералов и идентификаторов, дерево разбора, таблица ошибок. Далее происходит анализ дерева, с проходом по вершинам, с использованием таблиц литералов и идентификаторов. В случае возникновения ошибки, ошибка записывается в таблицу ошибок, включает информацию: код ошибки в таблице сообщений, номер строки. Пользователь может ознакомиться с ошибками, открыв протокол. Выполнение программы завершается.

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.

Выражения в языке BVD-2021 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 6.1 — Пример преобразования выражения в обратную польскую запись

Исходная строка	Результирующая строка	Стек
x+y*101b/(z-1b)		
+y*101b/(z-1b)	x	
y*101b/(z-1b)	x	+
*101b/(z-1b)	xy	+
101b/(z-1b)	xy	+*
/(z-1b)	xy101b	+*
(z-1b)	xy101b*	+/
z-1b)	xy101b *	+/(
-1b)	xy101b *z	+/(
1b)	xy101b *z	+/(-

Окончание таблицы 6.1

Исходная строка	Результирующая строка	Стек
)	xy101b *z1b	+/(-
	xy101b *z1b -	+/
	xy101b *z1b -/	+
	xy101b *z1b -/+	

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

6.2 Программная реализация обработки выражений

Во время семантического анализа, при успешном разборе выражения, функция семантического анализа вызывает функцию преобразования к обратной-польской записи, которая преобразует таблицу лексем.

6.3 Контрольный пример

В приложении Г приведена 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 BVD-2021 прошёл предыдущие компоненты транслятора без ошибок. Схема данного этапа изображена на рисунке 7.1.

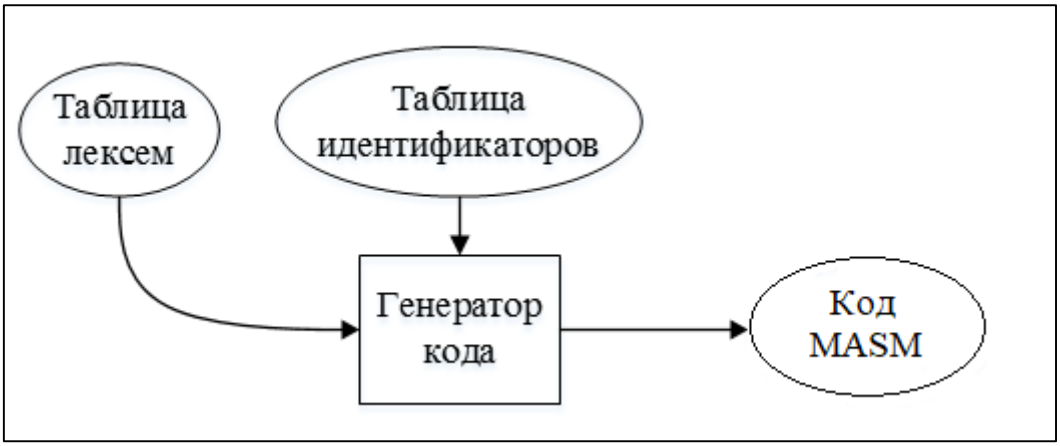

Рисунок 7.1 — Структура генератора кода

7.2 Представление типов данных в оперативной памяти

Элементы таблицы идентификаторов расположены в разных сегментах языка ассемблера — .data и .const. Идентификаторы языка BVD-2021 размещены в сегменте данных(.data). Литералы — в сегменте констант (.const). Соответствия между типами данных идентификаторов на языке BVD-2021 и на языке ассемблера приведены в таблице 7.1.

Таблица 7.1 — Соответствия типов идентификаторов языка BVD-2021 и языка Ассемблера

Тип идентификатора на языке BVD-2021	Тип идентификатора на языке ассемблера	Пояснение
byte	BYTE	Хранит целочисленный тип данных (1 байт).
str	DWORD	Хранит указатель на начало строки (4 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 BVD-2021 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
`declare byte Г;`	Ошибка 111:Недопустимый символ в исходном коде (-in)

8.2 Тестирование лексического анализатора

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

Таблица 8.2 — Тестирование лексического анализатора

Исходный код	Диагностическое сообщение
`declare str 134bv;`	Ошибка 120: Ошибка при распознавании лексемы
`declare str pow;`	Ошибка 125: Идентификатор уже занят в стандартной библиотеке

`major`
`{`
`output x;`
`declare str l;`
`l = “stroka;`
`return 0b;`
`};`
	Ошибка 121: Идентификатор не определен
	Ошибка 126: Закрывающая кавычка не найдена

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
`declare str x;` `major{...};`	Ошибка 600: Неверная структура программы.
`declare str x;` `x+1;`	Ошибка 601: Ошибочный оператор

Окончание таблицы 8.3

`declare str x;` `x = x x;`	Ошибка 602: Ошибка в выражении
`byte function fi(x,` `byte z){...};`	Ошибка 603: Ошибка в параметрах функции
`a = pow(byte);`	Ошибка 604: Ошибка в параметрах вызываемой функции
`declare byte a;` `if(a){...}`	Ошибка 605: Ошибка в условии

8.4 Тестирование семантического анализатора

На этапе синтаксического анализа могут возникнуть ошибки, описанные в пункте 5.3. Результат работы семантического анализатора приведены в таблице 8.4.

Таблица 8.4 — Тестирование семантического анализатора

Исходный код	Диагностическое сообщение
`declare byte x;` `declare byte x;`	Ошибка 610: Повторное объявление идентификатора
`declare byte function` `pow(byte a);`	Ошибка 611: Определенная функция не соответствует функциям стандартной библиотеки

`declare byte function`
`pow(byte a, byte b);`
`x = pow(x);`
	Ошибка 612: Формальные и фактические параметры функции не совпадают

`byte function haha(byte a){`
`declare str string;`
`declare byte b;`
`a = b + string;`
`}`
	Ошибка 613: Типы идентификаторов выражения не совпадают
	Ошибка 618: Переменная должна быть определена перед её использованием
	Ошибка 619: Последний оператор функции должен быть return

`declare str a;`
`a = “ggg”;`
`if(a < “neggg”){`
`declare byte a;`
`}`
	Ошибка 616: Типы сравниваемых значений должны быть byte
	Ошибка 617: Объявлять внутри if else и while блоков запрещено

`major`
`{`
`declare byte a;`
`declare byte function`
`abs(byte a);`
`a = abs;`
`declare str ds;`
`declare str dg;`
`ds = “g”;`
`dg = “b”;`
`ds = ds – dg;`
`return 1b;`
`};`
`major{...};`
	Ошибка 614: Функция должна иметь параметры
	Ошибка 615: К строкам нельзя применять операции, кроме конкатенации
	Ошибка 622: Главная функция major должна быть задана один раз

Заключение

В данном курсовом проекте были выполнены поставленные минимальные требования. В ходе работы было изучено много нового материала, а также закреплены знания, которые были получены ранее. 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, семантический анализатор, генератор кода и задачи, которые они должны решать. В итоге был получен примитивный язык программирования BVD-2021.

Окончательная версия языка BVD-2021 включает:

1) 2 типа данных;
2) Поддержка оператора вывода;
3) Возможность подключения и вызова функций стандартной библиотеки;
4) Наличие 5 арифметических операторов
5) Наличие условной конструкции и оператора цикла
6) Наличие 4 условных знаков
7) Возможность вызова функций

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.

Приложение А

Контрольный пример

```
str function ah(str a){
      declare str function strcat(str a, str b);
      a = strcat(a,"with the BVD-2021 programming language");
      output a;
      return a;
};

byte function addone(byte a, byte b){
      if(a < 0b){
      output "a < 0 ";
      }
      else{
      output "a >= 0 ";
      }
      a = a + b + 1b;
      return a;
};

major{
declare byte y;
declare byte x;
declare byte function pow(byte a, byte b);
declare byte function abs(byte a);
y = abs(10000110b);
x = pow(y, 10b)-1010b;
output y;
output x;
x = addone(x, 1b);
while(y < x){
y = y + 1b;
}
output y;
output x;
declare str a;
a = ah("You are working ");
output a;
return 0b;
};
```

Структуры данных лексического анализатора

```
namespace LT
{
      struct Entry
      {
            char lexema[LEXEMA_FIXSIZE];
```

```
            int   sn;
            int   idxTI;
            int  position;
            char idDataType = LEX_NULL;
            Entry();
            Entry(char l, int s, int i);
            Entry(char l, int s, int i, char dtype);
      };

      struct LexTable
      {
            int maxsize;
            int size;
            int funcIdInTI;
            Entry* table;
            bool SetEntryIdxTI(int idx);
            bool SetFuncIdInTI(int idf);
            int LastId();
      };

      Entry Putentry(char l, int s);
      Entry Putentry(char l, int s, char dtype);

      LexTable Create(
            int size
      );
      void Add(
            LexTable& lextable,
            Entry entry
      );
      Entry Getentry(LexTable& lextable, int n);


      void Delete(LexTable& lextable);
};


namespace IT
{
      enum class IDDATATYPE {
            NDEFD = 'n',
            STR = 's',
            BYTE = 'b',
      };
      enum class IDTYPE
      {
            NDEFT = 0,
            V = 1,
            F = 2,
            L = 3,
            P = 4,
```

```
		DF = 5,
		DFP = 6,
	};
	struct Entry
	{
		int		idxfirstLE;
		int		idxFuncIT = -1;
		unsigned char id[ID_MAXSIZE + 1] = "";
		IDDATATYPE iddatatype;
		IDTYPE		idtype;
		IDDATATYPE* parmsTypes = nullptr;
		bool	isdefined = false;
		Entry();
		Entry(int idxIT);
		Entry(int& idxfirst, unsigned char* iD, int funcId, IDDATATYPE&
iddat, IDTYPE& idtype);
		struct
		{
			short len;
			char* str;
		}	literalValue;
	};

	struct  IdTable
	{
		int maxsize;
		int size;
		Entry* table;
		int LastId();
	};
	bool PutLiteral(IdTable& idtable, LT::LexTable& lextable, unsigned
char* iD,
		IDDATATYPE iddat);
	bool Putentry(IdTable& idtable, LT::LexTable& lextable,  unsigned
char* iD,
		IDDATATYPE iddat, IDTYPE idt);
	bool PutMain(IdTable& idtable, LT::LexTable& lextable);
	IdTable Create(
		int size
	);
	int Add(
		IdTable& idtable,
		const Entry& entry
	);
	Entry Getentry(IdTable& idtable, int n);

	int IsId(IdTable& idtable, char id[ID_MAXSIZE]);

	void Delete(IdTable& idtable);
}
```

Таблица лексем

```
 0    tfi[0F](ti[1P]){
1     dtfi[2DF](ti[3DFP],ti[4DFP]);
2     i[1P]=i[2DF](i[1P],l("with the BVD-2021 programming language"));
3     pi[1P];
4     ri[1P];
5     };
6
7     tfi[6F](ti[7P],ti[8P]){
8     1(i[7P]<l(0b)){
9     pl("a < 0 ");
10    }
11    2{
12    pl("a >= 0 ");
13    }
14    i[7P]=i[7P]+i[8P]+l(1b);
15    ri[7P];
16    };
17
18    m[13F]{
19    dti[14V];
20    dti[15V];
21    dtfi[16DF](ti[17DFP],ti[18DFP]);
22    dtfi[19DF](ti[20DFP]);
23    i[14V]=i[19DF](l(10000110b));
24    i[15V]=i[16DF](i[14V],l(10b))-l(1010b);
25    pi[14V];
26    pi[15V];
27    i[15V]=i[6F](i[15V],l(1b));
28    w(i[14V]<i[15V]){
29    i[14V]=i[14V]+l(1b);
30    }
31    pi[14V];
32    pi[15V];
33    dti[26V];
34    i[26V]=i[0F](l("You are working "));
35    pi[26V];
36    rl(0b);
37    };
```

Таблица идентификаторов

```
 0: -1/ah(sF) : 0
1: 0/a(sP) : 0
2: 0/strcat(sDF) : 1
3: 0/a(sDFP) : 1
4: 0/b(sDFP) : 1
5: 0/!literal(sL = " with the BVD-2021 programming language") : 2
6: -1/addone(bF) : 7
7: 6/a(bP) : 7
```

```
8: 6/b(bP) : 7
9: 6/!literal(bL = 0b) : 8
10: 6/!literal(sL = "a < 0 ") : 9
11: 6/!literal(sL = "a >= 0 ") : 12
12: 6/!literal(bL = 1b) : 14
13: -1/!major(nF) : 18
14: 13/y(bV) : 19
15: 13/x(bV) : 20
16: 13/pow(bDF) : 21
17: 13/a(bDFP) : 21
18: 13/b(bDFP) : 21
19: 13/abs(bDF) : 22
20: 13/a(bDFP) : 22
21: 13/!literal(bL = 10000110b) : 23
22: 13/!literal(bL = 10b) : 24
23: 13/!literal(bL = 1010b) : 24
24: 13/!literal(bL = 1b) : 27
25: 13/!literal(bL = 1b) : 29
26: 13/a(sV) : 33
27: 13/!literal(sL = " You are working ") : 34
28: 13/!literal(bL = 0b) : 36
```

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;
		byte 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, byte 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);
		Rule getRule(short n); };
```

```
	const Greibach& getGreibach();
}

typedef std::vector<short> MFSTSTSTACK;
namespace MFST
{
	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];

		GRBALPHABET* lenta;
		short lenta_position;
		short nrule;
		short nrulechain;
		short lenta_size;
		GRB::Greibach grebach;
		LT::LexTable lexTable;
		MFSTSTSTACK st;
```

```
            std::vector<MfstState> storestate;

            Mfst();
            Mfst(LT::LexTable& plexTable, GRB::Greibach pgrebach);

            char* getCSt(char* buf);
            char* getCLenta(char* buf, short pos, short n = 25);
            char* getDiagnosis(short n, char* buf);
            Error::ErrorTable::Error getError();

            bool savestate();
            bool resetstate();
            bool push_chain(GRB::Rule::Chain chain);
            RC_STEP step();
            bool start();
            bool savedDiagnosis(RC_STEP prc_step);

            void printRules(std::ostream& stream);

            struct Deducation
            {
                  short size;
                  short* nrules;
                  short* nrulechains;

                  Deducation()
                  {
                        this->size = 0;
                        this->nrules = 0;
                        this->nrulechains = 0;
                  }
            } deducation;

            bool savededucation();
      };
}
```

Грамматика Грейбах:

```
      Greibach greibach(
            NS(START_SYMBOL_GBR), TS(BOTTOM_STACK_GBR),
            7,
            Rule(NS(START_SYMBOL_GBR), GRB_ERROR_SERIES + 0,
                  4,
                  Rule::Chain(5, TS('m'), TS('{'), NS('N'), TS('}'),
TS(';')),
                  Rule::Chain(11, TS('t'), TS('f'), TS('i'), TS('('),
NS('F'), TS(')'), TS('{'), NS('N'), TS('}'), TS(';'), NS('S')),
                  Rule::Chain(6, TS('m'), TS('{'), NS('N'), TS('}'), TS(';'),
NS('S')),
                  Rule::Chain(10, TS('t'), TS('f'), TS('i'), TS('('),
NS('F'), TS(')'), TS('{'), NS('N'), TS('}'), TS(';'))
```

```
          ),

          Rule(NS(SEQUENCE_OF_OPERATORS_GBR), GRB_ERROR_SERIES + 1,
               16,
               Rule::Chain(4, TS('d'), TS('t'), TS('i'), TS(';')),
               Rule::Chain(3, TS('r'), NS('E'), TS(';')),
               Rule::Chain(4, TS('i'), TS('='), NS('E'), TS(';')),
               Rule::Chain(8, TS('d'), TS('t'), TS('f'), TS('i'), TS('('),
NS('F'), TS(')'), TS(';')),
               Rule::Chain(3, TS('p'), NS('E'), TS(';')),
               Rule::Chain(5, TS('d'), TS('t'), TS('i'), TS(';'),
NS('N')),
               Rule::Chain(4, TS('r'), NS('E'), TS(';'), NS('N')),
               Rule::Chain(5, TS('i'), TS('='), NS('E'), TS(';'),
NS('N')),
               Rule::Chain(9, TS('d'), TS('t'), TS('f'), TS('i'), TS('('),
NS('F'), TS(')'), TS(';'), NS('N')),
               Rule::Chain(4, TS('p'), NS('E'), TS(';'), NS('N')),

               Rule::Chain(7, TS('1'), TS('('), NS('C'), TS(')'), TS('{'),
NS('N'), TS('}')),
               Rule::Chain(8, TS('1'), TS('('), NS('C'), TS(')'), TS('{'),
NS('N'), TS('}'), NS('N')),
               Rule::Chain(11, TS('1'), TS('('), NS('C'), TS(')'),
TS('{'), NS('N'), TS('}'), TS('2'), TS('{'), NS('N'), TS('}')),
               Rule::Chain(12, TS('1'), TS('('), NS('C'), TS(')'),
TS('{'), NS('N'), TS('}'), TS('2'), TS('{'), NS('N'), TS('}'), NS('N')),
               Rule::Chain(7, TS('w'), TS('('), NS('C'), TS(')'), TS('{'),
NS('N'), TS('}')),
               Rule::Chain(8, TS('w'), TS('('), NS('C'), TS(')'), TS('{'),
NS('N'), TS('}'), NS('N'))
          ),
          Rule(NS(EXPRESSION_GBR), GRB_ERROR_SERIES + 2,
               8,
               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(NS(MRULE_GBR), GRB_ERROR_SERIES + 2,
               10,
               Rule::Chain(2, TS('+'), NS('E')),
               Rule::Chain(2, TS('-'), NS('E')),
               Rule::Chain(2, TS('*'), NS('E')),
               Rule::Chain(2, TS('/'), NS('E')),
               Rule::Chain(2, TS('%'), NS('E')),
```

```
            Rule::Chain(3, TS('+'), NS('E'), NS('M')),
            Rule::Chain(3, TS('-'), NS('E'), NS('M')),
            Rule::Chain(3, TS('*'), NS('E'), NS('M')),
            Rule::Chain(3, TS('/'), NS('E'), NS('M')),
            Rule::Chain(3, TS('%'), NS('E'), NS('M'))
      ),
      Rule(NS(PARMS_GBR), GRB_ERROR_SERIES + 2,
            2,
            Rule::Chain(2, TS('t'), TS('i')),
            Rule::Chain(4, TS('t'), TS('i'), TS(','), NS('F'))
      ),
      Rule(NS(CONDITION_GBR), GRB_ERROR_SERIES + 5,
            12,
            Rule::Chain(3, TS('i'), TS('>'), TS('i')),
            Rule::Chain(3, TS('i'), TS('<'), TS('i')),
            Rule::Chain(3, TS('i'), TS('&'), TS('i')),
            Rule::Chain(3, TS('i'), TS('!'), TS('i')),
            //
            Rule::Chain(3, TS('i'), TS('>'), TS('l')),
            Rule::Chain(3, TS('i'), TS('<'), TS('l')),
            Rule::Chain(3, TS('i'), TS('&'), TS('l')),
            Rule::Chain(3, TS('i'), TS('!'), TS('l')),
            //
            Rule::Chain(3, TS('l'), TS('>'), TS('i')),
            Rule::Chain(3, TS('l'), TS('<'), TS('i')),
            Rule::Chain(3, TS('l'), TS('&'), TS('i')),
            Rule::Chain(3, TS('l'), TS('!'), TS('i'))
      ),

      Rule(NS(UNDER_EXPRESSION_GBR), GRB_ERROR_SERIES + 4,
            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

```
0   : S->tfi(F){N};S      0
4   : F->ti               1
8   : N->dtfi(F);N        2
13  : F->ti,F             3
16  : F->ti               4
20  : N->i=E;N            5
22  : E->i(W)             6
24  : W->i,W              7
26  : W->l                8
29  : N->pE;N             9
30  : E->i                10
32  : N->rE;              11
33  : E->i                12
37  : S->tfi(F){N};S      13
41  : F->ti,F             14
44  : F->ti               15
48  : N->1(C){N}2{N}N     16
50  : C->i<l              17
55  : N->pE;              18
56  : E->l                19
61  : N->pE;              20
62  : E->l                21
65  : N->i=E;N            22
67  : E->iM               23
68  : M->+E               24
69  : E->iM               25
70  : M->+E               26
71  : E->l                27
73  : N->rE;              28
74  : E->i                29
78  : S->m{N};            30
80  : N->dti;N           31
84  : N->dti;N           32
88  : N->dtfi(F);N       33
93  : F->ti,F            34
96  : F->ti              35
100 : N->dtfi(F);N       36
105 : F->ti              37
109 : N->i=E;N           38
111 : E->i(W)            39
113 : W->l               40
116 : N->i=E;N           41
118 : E->i(W)M           42
120 : W->i,W             43
122 : W->l               44
124 : M->-E              45
125 : E->l               46
127 : N->pE;N            47
128 : E->i               48
```

```
130 : N->pE;N             49
131 : E->i                50
133 : N->i=E;N            51
135 : E->i(W)             52
137 : W->i,W              53
139 : W->l                54
142 : N->w(C){N}N         55
144 : C->i<i              56
149 : N->i=E;             57
151 : E->iM               58
152 : M->+E               59
153 : E->l                60
156 : N->pE;N             61
157 : E->i                62
159 : N->pE;N             63
160 : E->i                64
162 : N->dti;N            65
166 : N->i=E;N            66
168 : E->i(W)             67
170 : W->l                68
173 : N->pE;N             69
174 : E->i                70
176 : N->rE;              71
177 : E->l                72
```

Приложение В

Ошибки семантического анализатора:

Пример 1

```
major
{
declare byte function pow(str p);
declare byte num;
num = pow(10b)
return 0b;
};
```

Ошибка 612: Формальные и фактические параметры функции не совпадают

Пример 2

```
major
{
declare byte num;
num = “s”;
output num;
return 0;
};
```

Ошибка 613 : Типы идентификаторов выражения не совпадают

Пример 3

```
major
{
declare byte x;
return x;
};
```

Ошибка 618: Переменная должна быть определена перед её использованием

Приложение Г

Программная реализация обработки выражений:

```
short GetPriority(char lex) {
           if (lex == LEX_LEFTHESIS || lex == LEX_RIGHTHESIS) return 0;
           if (lex == LEX_COMMA) return 1;
           if (lex == LEX_STAR || lex == LEX_DIRSLASH || lex == LEX_PEER)
return 2;
           if (lex == LEX_PLUS || lex == LEX_MINUS) return 3;
           return -1;
     }

     bool PriorityMoreOrEqual(char first, char second) {
           return GetPriority(first) >= GetPriority(second) ? true : false;
     }
     bool PolishNotation(int lextable_pos, LT::LexTable& lextable) {
           LT::LexTable table = LT::Create(0);
           char firststr[50];
           char secondstr[50];
           for (int poslex = lextable_pos; lextable.table[poslex].lexema[0]
!= LEX_SEMICOLON; poslex++) {
                firststr[poslex - lextable_pos] = lextable.ta-
ble[poslex].lexema[0];
                secondstr[poslex - lextable_pos] = '1';
           }
           short secstrcount = 0;
           stacker stack;
           short parmsCounter = 1;
           LT::Entry function_info;
           int pos;
           for (pos = lextable_pos;
                lextable.table[pos].lexema[0] != LEX_SEMICOLON; pos++) {
                if (lextable.table[pos].idxTI != -1) {
                     if (lextable.table[pos-1].idxTI != -1) return false;
                     if (lextable.table[pos + 1].lexema[0] == LEX_LEFTHE-
SIS) {
                          function_info = lextable.table[pos];
                          stack.push(LEX_FUNC_PN);
                          parmsCounter = 1;
                     }
                     else {
                          LT::Add(table, lextable.table[pos]);
                          secondstr[secstrcount++] = lextable.ta-
ble[pos].lexema[0];
                     }
                }
                else if (lextable.table[pos].lexema[0] == LEX_PLUS ||
                     lextable.table[pos].lexema[0] == LEX_MINUS ||
```

```
                    lextable.table[pos].lexema[0] == LEX_STAR ||
                    lextable.table[pos].lexema[0] == LEX_DIRSLASH ||
                    lextable.table[pos].lexema[0] == LEX_PERC ||
                    lextable.table[pos].lexema[0] == LEX_PEER  )
              {
                    if (GetPriority(lextable.table[pos - 1].lexema[0]) >=
1 &&
                          GetPriority(lextable.table[pos - 1].lexema[0]) <=
3 ||
                          lextable.table[pos - 1].lexema[0] == LEX_LEFTHE-
SIS
                          )
                          return false;
                    if (stack.isEmpty() || stack.getHead() == LEX_LEFTHE-
SIS) {}
                    else if (GetPriority(stack.getHead()) <= GetPrior-
ity(lextable.table[pos].lexema[0]))
                    {
                          while (!stack.isEmpty() && GetPriority(stack.get-
Head()) <= GetPriority(lextable.table[pos].lexema[0])) {
                                secondstr[secstrcount++] = stack.getHead();
                                LT::Add(table, LT::Putentry(stack.pop(),
0));
                          }
                    }
                    stack.push(lextable.table[pos].lexema[0]);
              }
              else if (lextable.table[pos].lexema[0] == LEX_COMMA) {
                    if (!stack.isEmpty()) {
                          while (stack.getHead() != LEX_LEFTHESIS) {
                                secondstr[secstrcount++] = stack.getHead();
                                LT::Add(table, LT::Putentry(stack.pop(),
0));
                          }
                    }
                    parmsCounter++;
              }
              else if (lextable.table[pos].lexema[0] == LEX_LEFTHESIS) {
                    stack.push(lextable.table[pos].lexema[0]);
              }
              else if (lextable.table[pos].lexema[0] == LEX_RIGHTHESIS) {
                    if (lextable.table[pos - 1].idxTI != -1 ||
                          GetPriority(lextable.table[pos - 1].lexema[0]) ==
0)
                    {
                          while (!stack.isEmpty() && stack.getHead() !=
LEX_LEFTHESIS) {
                                secondstr[secstrcount++] = stack.getHead();
                                LT::Add(table, LT::Putentry(stack.pop(),
0));
                          }
```

```
                        if (stack.isEmpty())
                              return false;
                        stack.pop();
                        if (!stack.isEmpty() && stack.getHead() ==
LEX_FUNC_PN) {
                              secondstr[secstrcount++] = LEX_ID;
                              secondstr[secstrcount++] = LEX_FUNC_PN;
                              secondstr[secstrcount++] = (char)((int)'0'
+ parmsCounter);
                              LT::Add(table, function_info);
                              LT::Add(table, LT::Putentry(stack.pop(),
0));

                        }
                  }
                  else
                        return false;
            }
            else
                  return false;
      }
      while (!stack.isEmpty()) {
            secondstr[secstrcount++] = stack.getHead();
            if (stack.getHead() == LEX_LEFTHESIS) return false;
            LT::Add(table, LT::Putentry(stack.pop(), 0));
      }
      int lex_override_position;
      int new_table_size = lextable_pos + table.size;
      for (lex_override_position = lextable_pos; lex_override_position
< new_table_size; lex_override_position++) {
            lextable.table[lex_override_position] = table.ta-
ble[lex_override_position - lextable_pos];
      }
      lextable.table[lex_override_position] = lextable.table[pos];
      for (lex_override_position++; lex_override_position <= pos;
lex_override_position++) {
            lextable.table[lex_override_position] =
LT::Putentry(LEX_NULL, 0);
      }
      secondstr[secstrcount] = '\0';
      LT::Delete(table);
      return true;
}
```

Приложение Д

Результат генерации кода:

```
.586
.model flat, stdcall
includelib kernel32.lib
includelib ../Debug/MASMStaticLib.lib

STD_OUTPUT            EQU - 11
ExitProcess                PROTO : DWORD
GetStdHandle          PROTO : DWORD;
WriteConsoleA         PROTO : DWORD, : DWORD, : DWORD, : DWORD, : DWORD
SetConsoleTitleAPROTO : DWORD
byte_to_char               PROTO : SBYTE , : DWORD
_pow                       PROTO : SBYTE, : SBYTE
_abs                       PROTO : SBYTE
_strcat                         PROTO : DWORD, : DWORD
__push                          PROTO
__pop                      PROTO
fill_str                   PROTO : DWORD,: DWORD
find_len                   PROTO
_ah PROTO : DWORD
_addone PROTO : SBYTE,: SBYTE

.const
newStr BYTE 10, 13
newStr_len BYTE 2
zeroEx BYTE "Division by zero", 0
literal5 BYTE "with the BVD-2021 programming language", 0
literal9 SBYTE 0b
literal10 BYTE "a < 0 ", 0
literal11 BYTE "a >= 0 ", 0
literal12 SBYTE 1b
literal21 SBYTE -0000110b
literal22 SBYTE 10b
literal23 SBYTE 1010b
literal24 SBYTE 1b
literal25 SBYTE 1b
literal27 BYTE "You are working ", 0
literal28 SBYTE 0b

.data
buffer BYTE 10 dup(0)
bufferstr BYTE 255 dup(0)
bufferstrSymbol dword 0
consoleOutHandledword ?
_13y SBYTE 0b
_13x SBYTE 0b
```

```
_13a DWORD 0
str0 byte 255 dup(0)
str1 byte 255 dup(0)
str2 byte 255 dup(0)
str3 byte 255 dup(0)
str4 byte 255 dup(0)
str5 byte 255 dup(0)
str6 byte 255 dup(0)

.code
_ah PROC  a0: DWORD
invoke _strcat,  a0, offset literal5
invoke fill_str, offset str0, eax
mov a0,eax
mov eax, a0
invoke fill_str, offset str1, eax
call find_len

invoke WriteConsoleA, consoleOutHandle, eax, edx, 0, 0
invoke WriteConsoleA, consoleOutHandle, offset newStr , newStr_len, 0, 0
mov eax, a0
invoke fill_str, offset str2, eax

ret
_ah ENDP
_addone PROC  a6: SBYTE, b6: SBYTE
mov ah, a6
mov al, literal9
.if ah < al
mov eax, offset literal10
invoke fill_str, offset str3, eax
call find_len

invoke WriteConsoleA, consoleOutHandle, eax, edx, 0, 0
invoke WriteConsoleA, consoleOutHandle, offset newStr , newStr_len, 0, 0
.else
mov eax, offset literal11
invoke fill_str, offset str4, eax
call find_len

invoke WriteConsoleA, consoleOutHandle, eax, edx, 0, 0
invoke WriteConsoleA, consoleOutHandle, offset newStr , newStr_len, 0, 0
.endif
mov al, a6
call __push
mov al, b6
call __push

mov  eax, 0b
call __pop
mov       bl, al
```

```
call __pop
add        al, bl
jno        isOk0
mov        al, 0b
isOk0:
call __push
mov al, literal12
call __push

mov  eax, 0b
call __pop
mov        bl, al
call __pop
add        al, bl
jno        isOk1
mov        al, 0b
isOk1:
call __push
call __pop
 mov a6, al
mov al, a6
call __push
call __pop
ret
_addone ENDP
main PROC
invoke     GetStdHandle, STD_OUTPUT
mov  consoleOutHandle, eax
mov al, literal21
call __push
call __pop
push eax
call _abs
call __push
call __pop
 mov _13y, al
mov al, _13y
call __push
mov al, literal22
call __push
call __pop
push eax
call __pop
push eax
call _pow
call __push
mov al, literal23
call __push

mov  eax, 0b
call __pop
```

```
mov        bl, al
call __pop
sub        al, bl
jno        isOk2
mov        al, 0b
isOk2:
call __push
call __pop
 mov _13x, al
mov al, _13y
call __push
call __pop
invoke     byte_to_char, al, offset buffer
invoke WriteConsoleA, consoleOutHandle, offset buffer, sizeof buffer, 0, 0
invoke WriteConsoleA, consoleOutHandle, offset newStr , newStr_len, 0, 0

mov al, _13x
call __push
call __pop
invoke     byte_to_char, al, offset buffer
invoke WriteConsoleA, consoleOutHandle, offset buffer, sizeof buffer, 0, 0
invoke WriteConsoleA, consoleOutHandle, offset newStr , newStr_len, 0, 0

mov al, _13x
call __push
mov al, literal24
call __push
call __pop
push eax
call __pop
push eax
call _addone
call __push
call __pop
 mov _13x, al
mov ecx, 2
while0:
mov ah, _13y
mov al, _13x
.if ah < al
mov ecx, 2
.else
jmp while0end
.endif
mov al, _13y
call __push
mov al, literal25
call __push

mov  eax, 0b
call __pop
```

```
mov        bl, al
call __pop
add        al, bl
jno        isOk3
mov        al, 0b
isOk3:
call __push
call __pop
 mov _13y, al
loop while0
while0end:
mov al, _13y
call __push
call __pop
invoke     byte_to_char, al, offset buffer
invoke WriteConsoleA, consoleOutHandle, offset buffer, sizeof buffer, 0, 0
invoke WriteConsoleA, consoleOutHandle, offset newStr , newStr_len, 0, 0

mov al, _13x
call __push
call __pop
invoke     byte_to_char, al, offset buffer
invoke WriteConsoleA, consoleOutHandle, offset buffer, sizeof buffer, 0, 0
invoke WriteConsoleA, consoleOutHandle, offset newStr , newStr_len, 0, 0

invoke _ah,  offset literal27
invoke fill_str, offset str5, eax
mov _13a,eax
mov eax, _13a
invoke fill_str, offset str6, eax
call find_len

invoke WriteConsoleA, consoleOutHandle, eax, edx, 0, 0
invoke WriteConsoleA, consoleOutHandle, offset newStr , newStr_len, 0, 0
mov al, literal28
call __push
call __pop
ret

invoke ExitProcess, 0
main ENDP
end main
```