МИНИСТЕРСТВО ОБРАЗОВАНИЯ РЕСПУБЛИКИ БЕЛАРУСЬ

Учреждение образования «БЕЛОРУССКИЙ
ГОСУДАРСТВЕННЫЙ ТЕХНОЛОГИЧЕСКИЙ УНИВЕРСИТЕТ»

Факультет Информационных технологий
Кафедра Программной инженерии
Специальность 6-05-0612-01 Программная инженерия.
Профилизация: Программное обеспечение информационных технологий

ПОЯСНИТЕЛЬНАЯ ЗАПИСКА
К КУРСОВОМУ ПРОЕКТУ НА ТЕМУ:

«Разработка компилятора ZMI-2025»

Выполнил студент 2-10 _____ Загнетова Мария Игоревна
курс, группа подпись (Ф.И.О. студента)

Руководитель проекта _____ асс. Волчек Д. И.
(учен. степень, звание, должность, подпись, Ф.И.О. руководителя)

Заведующий кафедрой _____ к.т.н., доц. Смелов В.В.
(учен. степень, звание, должность, подпись, Ф.И.О.)

Консультанты _____ асс. Волчек Д. И.
(учен. степень, звание, должность, подпись, Ф.И.О.)

Курсовой проект защищен с оценкой _____

Минск 2025

Содержание

Введение

Целью курсового проекта является создание своего языка программирования ZMI-2025 и реализация компилятора. Написание транслятора будет осуществляться на языке C++, а код на языке ZMI-2025 будет транслироваться в язык Assembler.

Компилятор ZMI-2025 состоит из следующих частей:

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

На основе поставленной цели курсового проекта, были выявлены следующие задачи:

- разработка спецификации языка ZMI-2025;
- разработка лексического анализатора;
- разработка синтаксического анализатора;
- разработка семантического анализатора;
- разбор арифметических выражений;
- разработка генератора кода;
- тестирование транслятора.

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

1 Спецификация языка программирования

1.1 Характеристика языка программирования

Язык программирования ZMI-2025 является процедурным, универсальным, строго типизированным, не объектно-ориентированным, компилируемым.

Поддерживает 4 типа данных: целочисленный (byte), строковый (text), символьный (char) и логический (bool). В стандартной библиотеке языка программирования доступны расширенный набор функции: для лексикографического сравнения строк stcmp(text s1, text s2), преобразования строки в число strtoint(text s), вычисление длины строки strle(text s), вычисление модуля числа mabs(byte n) и генерация случайных чисел rnd(byte max).

1.2 Определение алфавита языка программирования

При написании программы на языке ZMI-2025 используется таблица символов Windows-1251.

Исходный код ZMI-2025 может содержать символы-сепараторы, символы операций, символы латинского алфавита: прописные и строчные, цифры десятичной системы счисления от 0 до 9, символы кириллицы [а…я], [А…Я] разрешены только в строковых литералах и комментариях.

В качестве сепараторов и специальных символов используются: {} [ ] ( ) , ; : + - / * > < ! “пробел”.

Для записи инструкций языка используются символы: [a…z].

1.3 Применяемые сепараторы

Для того, чтобы разделить инструкции языка, необходимо использовать сепараторы. Примеры используемых в языке программирования ZMI-2025 сепараторов приведены в таблице 1.1.

Таблица 1.1. – Сепараторы

Сепаратор	Назначение
пробел	Разделитель цепочек. Допускается везде кроме идентификаторов и ключевых слов
{ … }	Ограничители блоков кода (функций, box, конструкции check)
(…)	Ограничители списков параметров, а также изменение приоритета арифметических операций
[…]	Ограничители комментариев
,	Разделитель элементов списка параметров
+ - * / %	Арифметические операции
== != < > <= >=	Операции сравнения
;	Признак конца инструкции
<	Оператор вывода данных (в конструкции show) и операция сравнения
=	Оператор присваивания

В контексте языка программирования ZMI-2025 применяются различные сепараторы с разной логической нагрузкой.

1.4 Применяемые кодировки

Для написания исходного кода программы на языке ZMI-2025 используется кодировка Windows-1251. На рисунке 1.1 представлена таблица кодировки Windows-1251.

	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 – Таблица кодировки Windows-1251

Данная кодировка полностью соответствует всем требованиям для формирования алфавита языка.

1.5 Типы данных

Тип данных – это понятие, которое определяет набор значений, которые может принимать переменная или выражение, а также операции, которые могут быть выполнены над этими значениями.

В языке ZMI-2025 реализованы четыре типа данных. А именно целочисленный (byte), строковый (text), символьный (char) и логический (bool). Описание типов приведено в таблице 1.2.

Таблица 1.2 – Типы данных языка ZMI-2025

Тип данных	Характеристика
1	2
byte	Целочисленный тип данных. Используется для работы с числовыми значениями. В памяти занимает 1 байт. Диапазон значений: от -128

Продолжение таблицы 1.2

1	2
	до 127. Задание значения возможно в десятичной, двоичной (0b) и шестнадцатеричной (0x) системах счисления. Инициализация по умолчанию: значение 0. Поддерживаемые операции: + (бинарный) – оператор сложения; - (бинарный) – оператор вычитания; * (бинарный) – оператор умножения; / (бинарный) – оператор деления; % (бинарный) – оператор определения остатка от деления; = (бинарный) – оператор присваивания. В качестве условия условного оператора поддерживаются следующие логические операции: > (бинарный) – оператор «больше»; < (бинарный) – оператор «меньше»; <= (бинарный) – оператор «меньше или равно»; >= (бинарный) – оператор «больше или равно»; == (бинарный) – оператор «равенства»; != (бинарный) – оператор «неравенство»;
text	Строковый тип данных . Вся строка занимает n + 1 байт, где n – количество символов в строке и + 1 для символа конца строки. Максимальная длина строки 255 символов. Каждый символ занимает 1 байт, символы расположены в памяти друг за другом. Строка оканчивается символом конца строки. Инициализация по умолчанию: \0. Операции над данными строкового типа: присваивание строковому идентификатору значения другого строкового идентификатора или строкового литерала.
char	Символьный тип данных. Занимает 1 байт. Используется для хранения одиночного символа в кодировке Windows-1251.
bool	Логический тип данных. Принимает значения true или false. В памяти хранится как 1 байт (1 или 0).

Представленные типы данных позволяют выполнять операции сложения, вычитания, умножения, деления с остатком и без, а также операции сравнения.

1.6 Преобразование типов данных

В языке программирования ZMI-2025 реализована строгая статическая типизация. Неявное преобразование между строковыми и числовыми типами запрещено, что повышает надежность кода.

Однако поддерживаются следующие виды преобразований:

–неявное совместимость, а именно типы byte и char считаются совместимыми, так как оба имеют размер 1 байт. Допускается использование символьных литералов в арифметических выражениях;

– явное преобразование (для преобразования строкового значения в целочисленное в стандартной библиотеке реализована функция strtoint(text s).

–обратное преобразование (числа в строку) выполняется автоматически только при выводе данных оператором show).

Подобный подход значительно расширяет функционал языка для различных операций и вычислений.

1.7 Идентификаторы

Идентификаторы используются для наименования функций и их параметров, переменных. Имена идентификаторов могут содержать символы латинского алфавита (прописные и строчные), цифры и символ подчеркивания (_). Идентификатор должен начинаться с буквы. Максимальная длина идентификатора составляет 32 символа.

Идентификаторы не могут содержать символы кириллицы. Использование русских букв в именах переменных или функций приведет к ошибке лексического анализа. Кириллица допускается исключительно внутри строковых литералов и в комментариях языка.

В языке предусмотрен список зарезервированных слов, которые не могут использоваться в качестве идентификаторов. К ним относятся ключевые слова языка (box, byte и т.д.) и некоторые команды ассемблера (eax, stack, data), чтобы избежать конфликтов при генерации объектного кода.

Представление идентификатора в РБНФ (Расширенная форма Бэкуса-Наура):

- <идентификатор> :: = <буква> { <буква> | <цифра> | };
- <буква>::= a | b | ... | z | A | B | ... | Z ;
- <цифра> ::= 0 | 1 | ... | 9;

Примеры правильных идентификаторов: result, my_var, power_func.

Примеры неправильных идентификаторов: 2num (начинается с цифры), my-var (содержит недопустимый символ), номер (на кириллице).

1.8 Литералы

Литерал – это запись в исходном коде программы, представляющая собой фиксированное значение.

С помощью литералов осуществляется инициализация переменных и использование фиксированных значений (литералов). Все литералы являются rvalue. Типы литералов языка ZMI-2025 представлены в таблице 1.3.

Таблица 1.3 – Описание литералов

Литералы	Пояснение
1	2
Целочисленный (десятичный)	Последовательность цифр [0-9]. Значение должно находиться в диапазоне от -128 до 127 (так как базовый тип byte занимает 1 байт).
Целочисленный (шестнадцатеричный)	Начинается с префикса 0x. Может состоять из цифр [0-9] и символов [A-F, a-f]. Максимальное значение 0xFF (255). Пример: 0x1A.

Продолжение таблицы 1.3

1	2
Целочисленный (двоичный)	Начинается с префикса 0b. Может состоять только из цифр 0 и 1. Максимальное значение 8 бит (до 0b11111111). Пример: 0b101.
Строковый	Последовательность символов, заключенная в двойные кавычки ("). Может содержать символы латинского алфавита, кириллицы, цифры и спецсимволы. Максимальная длина строки — 255 символов. Внутри строкового литерала допускается использование одинарных кавычек, но запрещено использование двойных.
Символьный	Одиночный символ, заключенный в одинарные кавычки ('). Пример: 'z', 'Я'. В памяти хранится как код символа (1 байт).
Логический	Представлен зарезервированными ключевыми словами true (истина) и false (ложь).

Ограничения на строковые литералы языка ZMI-2025: Строковые литералы ограничиваются двойными кавычками. Внутри литерала не допускается использование двойных кавычек (так как они служат терминатором строки), однако допускается использование одинарных кавычек. Экранирование символов не поддерживается.

1.9 Объявление данных

Для объявления переменной указывается тип данных и имя идентификатора. Особенностью языка ZMI-2025 является строгое разделение объявления и инициализации: инициализация переменной при её объявлении не допускается. Значение присваивается отдельной инструкцией.

Пример объявления числового типа с инициализацией:

- byte num1;
- num1 = -1;
- byte num2;
- num2 = 0xFF.

Пример объявления переменной символьного типа с инициализацией:

- Text str1;
- str1= "hello world".

Для объявления функций и процедур используется ключевое слово proc, перед которым указывается тип функции. Далее обязателен список параметров и тело функции.

Пример объявления функции:

- byte proc myFunc(byte a, byte b) { ... }.

1.10 Инициализация данных

В языке ZMI-2025 инициализация переменной непосредственно при её объявлении не поддерживается. Инициализация осуществляется путем присваивания значения уже объявленной ранее переменной.

При объявлении переменной компилятор автоматически присваивает ей значение по умолчанию:

– Значение 0 для целочисленного типа данных (byte).

– Пустая строка для строкового типа данных (text).

Способы работы с переменными языка программирования ZMI-2025 представлены в таблице 1.4.

Таблица 1.4 – Способы инициализации переменных

Вид инициализации	Примечание
<тип данных> <идентификатор>;	Объявление переменной с автоматической инициализацией (byte становится 0, text становится пустой строкой).
<идентификатор> = <значение>;	Явное присваивание начального значения (инициализация). Должно выполняться отдельной строкой после объявления.

Объектами-инициализаторами могут быть литералы, идентификаторы или результаты функций.

1.11 Инструкции языка

Инструкции языка ZMI-2025 представлены в таблице 1.5.

Таблица 1.5 – Инструкции языка ZMI-2025

Инструкция	Синтаксис
Объявление переменной	<тип данных> <идентификатор>;
Возврат из функции	ret <выражение>;
Вывод данных	show < <выражение>; Осуществляет вывод значения выражения в стандартный поток вывода с автоматическим переходом на новую строку.
Вызов функции	<идентификатор функции> (<список параметров>); Список параметров может быть пустым.
Присваивание	<идентификатор> = <выражение>; Выражением может быть идентификатор, литерал или вызов функции соответствующего типа. Для типа byte выражение может быть дополнено арифметическими операциями (+, -, *, /) с использованием скобок. Для типа text выражение может быть только идентификатором или строковым литералом.

Условный оператор (выбор)	check (<выражение>) {
	is <литерал>: {
	<операторы>

Продолжение таблицы 1.5

1	2

	}
	else: {
	<операторы>
	}
	}

Таким образом, программные инструкции языка представляют собой базовый функционал, позволяющий решать задачи различного уровня.

1.12 Операции языка

В языке ZMI-2025 предусмотрены следующие операции с данными. Приоритетность операций умножения, деления и остатка от деления выше приоритета операций сложения и вычитания.

Операции языка представлены в таблице 1.6.

Таблица 1.6 – Операции языка ZMI-2025

Тип оператора	Оператор
Арифметические	+ – сложение - – вычитание * – умножение / – деление без остатка % – остаток от деления
Присваивание	= –присваивает значение справа переменной слева
Сравнения	== – равно != – не равно < – меньше > – больше <= – меньше или равно >= – больше или равно
Специальный	< – ператор направления потока (используется только в инструкции show)

Если у операций одинаковый приоритет, то первой будет выполнена операция, стоящая левее. С помощью круглых скобок может быть изменен приоритет операций.

1.13 Выражения и их вычисления

Вычисление выражений – одна из важнейших задач транслятора. В языке ZMI-2025 выражения могут быть арифметическими (для типа byte) или строковыми (для типа text).

Правила составления выражений:

– Допускается использование круглых скобок ( ) для явного изменения приоритета операций.

– Между двумя операндами должен находиться ровно один оператор (конструкции вида a ++ b запрещены).

– В качестве операнда может выступать литерал, идентификатор переменной или вызов функции.

– Тип возвращаемого значения функции внутри выражения должен быть совместим с остальными операндами (число с числами, строка со строками).

– Арифметические выражения вычисляются с учетом знаковости типа byte (диапазон -128..127).

Перед этапом генерации кода все выражения (включая условия в операторе check и аргументы функций) преобразуются в Обратную Польскую Запись. Это позволяет линеаризовать порядок вычислений, избавиться от скобок и использовать стековую модель при генерации ассемблерного кода.

1.14 Конструкции языка

Программа на языке ZMI-2025 организуется в виде функций пользователя и главной функции. Программные конструкции языка представлены в таблице 1.7.

Таблица 1.7 – Программные конструкции языка ZMI-2025

Конструкция	Реализация

Точка входа)	box
	{
	…
	}

Функция	<тип данных> proc <идентификатор> (<тип> <идентификатор>, ...)
	{
	…
	ret <выражение>;
	}

Процедура (без возврата)	void proc <идентификатор> (<тип> <идентификатор>, ...) {… ret; }

Условный оператор (выбор из множества)	check (<выражение>)
	{
	is <литерал>:
	{ ... }
	else:
	{ ... }
	}
	<выражение> вычисляется один раз. Его значение последовательно сравнивается с литералами в ветках is. При совпадении выполняется соответствующий блок. Если совпадений нет, выполняется блок else..

Таким образом, программные конструкции языка представляют собой базовый функционал, позволяющий решать задачи различного уровня.

1.15 Область видимости идентификаторов

В языке ZMI-2025 поддерживается только локальная область видимости.

Правила видимости:

–Переменные, объявленные внутри функции, процедуры или блока box, доступны только в пределах этого блока.

–Параметры функции видны только внутри тела этой функции.

–Переменные, объявленные в одной функции, недоступны для обращения из другой функции.

Глобальные переменные не предусмотрены синтаксисом языка.

1.16 Семантические проверки

В языке программирования ZMI-2025 выполняются следующие семантические проверки:

– Наличие функции box (точки входа в программу).

– Единственность точки входа.

– Использование идентификаторов (переменных и функций) только после их объявления.

–Проверка соответствия типов данных при присваивании (запрещено присваивать строку числу и наоборот).

–Проверка совместимости типов операндов в арифметических выражениях (операции +, -, *, / разрешены только для числовых типов).

–Проверка соответствия типа возвращаемого значения (инструкция **ret**) объявленному типу функции.

–Правильность вызова функций: соответствие количества и типов передаваемых аргументов объявлению функции.

–Контроль диапазонов: проверка выхода за границы допустимых значений для числовых литералов (byte: -128..127) и длины строковых литералов (до 255 символов).

–Полнота условного оператора: обязательное наличие ветки else в конструкции check.

–Запрет деления на ноль (при использовании целочисленных литералов).

Семантические проверки необходимы для нахождения логических ошибок, обеспечения правильного смысла кода, а также для согласования разных частей программы.

1.17 Распределение оперативной памяти на этапе выполнения

Исполняемый файл, полученный в результате трансляции, использует три основные области памяти:

Сегмент констант (.const): Сюда заносятся все строковые и числовые литералы. Данные в этой области доступны только для чтения.

Сегмент данных (.data): Здесь размещаются все переменные программы. Особенность реализации: Переменные, объявленные внутри функций, физически размещаются в глобальной статической памяти (сегмент .data). Однако, благодаря механизму префиксов (добавление имени функции к имени переменной на этапе лексического анализа), на уровне исходного кода обеспечивается их строгая локальность и уникальность.

Стек (.stack): Используется для двух целей:Передача фактических параметров при вызове функций (согласно соглашению stdcall).Хранение промежуточных результатов вычислений (так как генератор кода использует стековую модель для обработки выражений из польской записи).

1.18 Стандартная библиотека и ее состав

Язык программирования ZMI-2025 предоставляет доступ к стандартной статической библиотеке **InLib.lib**, которая подключается на этапе компоновки (Linker). Библиотека написана на языке C++ и обеспечивает выполнение математических операций, работу со строками и генерацию случайных чисел. Внутренние функции вывода (outnum, outstr, newline) вызываются транслятором автоматически при использовании оператора show.

Функции, доступные пользователю для вызова, представлены в таблице 1.8

Таблица 1.8 – Стандартная библиотека языка ZMI-2025

Функция	Описание
byte proc strle(text s)	Возвращает длину строки s. Результат ограничен диапазоном типа byte (до 127 символов).
byte proc stcmp(text s1, text s2)	Выполняет лексикографическое сравнение двух строк. Возвращает **0**, если строки равны. Возвращает отличное от нуля значение, если строки не равны.
byte proc strtoint(text s)	Преобразует строку, содержащую запись числа, в целочисленное значение типа byte.
byte proc mabs(byte n)	Возвращает абсолютное значение (модуль) числа n.
byte proc rnd(byte max)	Генерирует псевдослучайное целое число в диапазоне от 0 до max (не включая само значение max).

Представленные функции стандартной библиотеки позволяют высчитывать длину строки, сравнивать две строки, преобразование строки в число, возвращение модуля, а также генерация псевдослучайного числа.

1.19 Ввод и вывод данных

Ввод данных с клавиатуры в текущей версии языка не реализован. Для вывода данных используется оператор show <. Он является универсальным: транслятор автоматически определяет тип выводимого выражения (число или строка) и генерирует соответствующий вызов библиотечной функции. После вывода значения автоматически выполняется переход на новую строку.

Служебные функции стандартной библиотеки, используемые

генератором кода для реализации вывода, представлены в таблице 1.9.

Таблица 1.9 – Служебные функции вывода

Функция на языке C++	Описание
int __stdcall outnum(int value)	Выводит в стандартный поток значение целочисленного выражения.
int __stdcall outstr(char* value)	Выводит в стандартный поток строковый литерал или значение строковой переменной. Включает поддержку кириллицы (локаль Russian).
int __stdcall newline()	Выводит символ конца строки, обеспечивая переход на следующую строку после каждого оператора show.

Представленные в таблице функции значительно упрощают работу функций стандартной библиотеки.

1.20 Точка входа

В языке ZMI-2025 каждая программа должна содержать специальный блок box (аналог функции main). Именно с первой инструкции этого блока начинается последовательное выполнение команд программы.

1.21 Препроцессор

Отдельный препроцессор в языке ZMI-2025 не предусмотрен. Обработка комментариев (текст, заключенный в квадратные скобки [...]) выполняется на этапе лексического анализатора (сканера), который игнорирует содержимое скобок при формировании потока лексем.

1.22 Соглашения о вызове

В генерируемом коде используется стандартное соглашение о вызовах stdcall:

- Параметры функции передаются через стек в порядке справа налево.
- Очистку стека от аргументов производит вызываемая функция (callee) при возврате управления.
- Возвращаемое значение функции помещается в регистр EAX (или AL для однобайтовых значений).

1.23 Объектный код

Результатом работы транслятора является файл с расширением .asm, содержащий исходный код на языке ассемблера (синтаксис MASM, архитектура x86). Для получения исполняемого файла (.exe) данный код впоследствии обрабатывается ассемблером и компоновщиком.

1.24 Классификация сообщений транслятора

Генерируемые транслятором сообщения определяют степень его информативности. Сообщения транслятора ZMI-2025 предоставляют полную информацию о месте и характере допущенной ошибки. При возникновении любой ошибки процесс трансляции прерывается.

Диапазоны кодов ошибок транслятора приведены в таблице 1.10.

Таблица 1.10 – Классификация ошибок

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

Каждая ошибка также содержит информацию о том, на каком этапе выполнения она была вызвана.

1.25 Контрольный пример

Исходный код контрольного примера представлен в приложении А.

2. Структура транслятора

2.1 Компоненты транслятора, их назначение и принципы взаимодействия

Транслятор – это программа, которая преобразует исходный код на одном языке программирования в исходный код на другом языке программирования.

На рисунке 2.1 представлена структура транслятора.

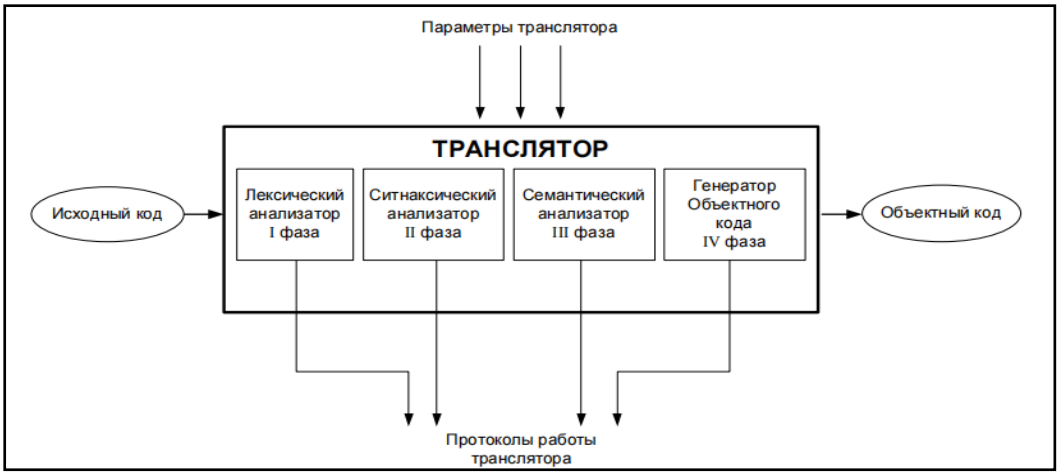

Рисунок 2.1 – Структура транслятора

Этапы трансляции выполняются последовательно. Каждый этап имеет свои входные и выходные данные, которые передаются на следующую фазу.

Первой фазой трансляции является лексический анализ. Принимает на вход исходный код программы. Задача лексического анализатора — распознать лексемы, удалить комментарии и пробельные символы, и сформировать две основные таблицы: таблицу лексем и таблицу идентификаторов.

Синтаксический анализ представляет собой вторую часть работы транслятора. Входом является таблица лексем. Задачей синтаксического анализатора является проверка последовательности лексем на соответствие правилам грамматики языка ZMI-2025 и построение дерева разбора.

Третьей фазой является семантический анализ. Проверяет смысловую корректность программы: соответствие типов данных, объявление идентификаторов, правильность вызова функций и отсутствие дубликатов. Использует таблицу идентификаторов и таблицу лексем.

Четвертой фазой является построение польской записи. Промежуточный этап, на котором арифметические выражения, логические условия и вызовы функций преобразуются в обратную польскую запись (ПОЛИЗ). Это необходимо для линеаризации выражений и упрощения их вычисления в стековой машине.

Последний этап трансляции представляет собой генерацию кода. На вход генератора подается таблица лексем (уже преобразованная в ПОЛИЗ) и

таблица идентификаторов. На их основе генерируется результирующий файл с кодом на языке ассемблера (MASM).

2.2 Перечень входных параметров транслятора

Для запуска транслятора и управления выводом результатов работы используются параметры командной строки. Перечень параметров языка программирования ZMI-2025 приведен в таблице 2.1.

Таблица 2.1 – Входные параметры транслятора языка ZMI-2025

Входной параметр	Описание параметра	Значение по умолчанию
-in:<путь к in-файлу>	Файл с исходным кодом на языке ZMI-2025 , имеющий расширение .txt	Не предусмотрено (обязательный параметр)
-log:<путь к log-файлу>	Файл журнала для вывода протоколов работы всех анализаторов и сообщений об ошибках.	Значение по умолчанию: <имя in-файла>.log
-out:<путь к out-файлу>	Файл, содержащий таблицу лексем (в том числе после преобразования в ПОЛИЗ) для отладки.	Значение по умолчанию: <имя in-файла>.out

Таблица 2.1 содержит параметры, которые принимает транслятор языка ZMI-2025 в качестве входных данных для создания файлов, содержащих результаты работы лексического, синтаксического и семантического анализаторов.

2.3 Протоколы, формируемые транслятором и их содержимое

В ходе работы программы формируются выходные файлы, содержащие результаты работы различных этапов трансляции. В таблице 2.2 приведен перечень формируемых протоколов и их содержимое.

Таблица 2.2 – Протоколы, формируемые транслятором языка ZMI-2025

Формируемый протокол	Описание выходного протокола
1	2
Файл журнала, заданный параметром "-log:"	Основной протокол работы транслятора. Содержит:параметры запуска и время старта, исходный текст программы, таблицу идентификаторов (типы данных и значения литералов), протокол работы синтаксического анализатора (дерево разбора/цепочку правил), сообщения об ошибках.
Файл журнала, заданный параметром "-out:"	Служебный файл для анализа промежуточных представлений. Содержит:таблицу лексем (поток лексем, сформированный лексическим анализатором), результат преобразования выражений в Обратную Польскую Запись.

Продолжение таблицы 2.2

1	2
Выходной файл, с расширением ".asm"	Результат успешной трансляции – файл, содержащий исходный код программы на языке ассемблера MASM. Генерируется только в случае отсутствия ошибок.

При возникновении критической ошибки на любом этапе работы анализаторов, транслятор прекращает свою работу. Подробная информация об ошибке записывается в log-файл и выводится в консоль. Генерация кода в файл .asm в таком случае не выполняется.

3 Разработка лексического анализатора

3.1 Структура лексического анализатора

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

Структурная схема лексического анализатора языка ZMI-2025 представлена на рисунке 3.1.

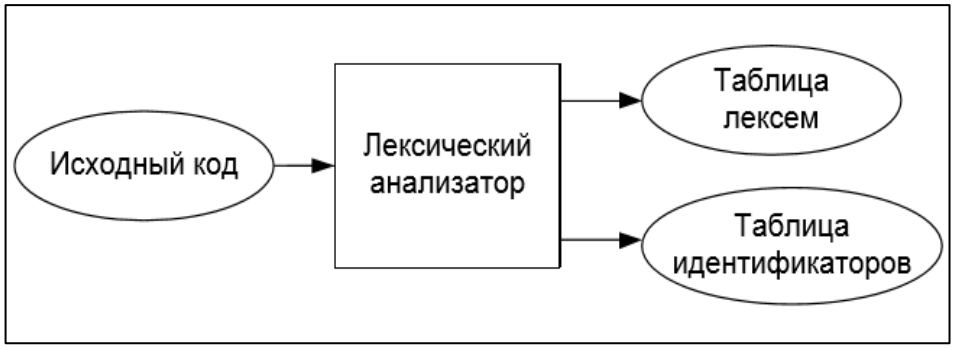

Рисунок 3.1 – Структурная схема лексического анализатора

Примеры лексических единиц включают идентификаторы, числа, символы операций и ключевые слова. Лексический анализатор преобразует исходный текст, заменяя лексические единицы их внутренним представлением - лексемами. Это создает промежуточное представление исходной программы. Каждой лексеме присваивается тип, и она записывается в таблицу лексем. Если лексема является идентификатором или литералом, она также заносится в таблицу идентификаторов.

Функции лексического анализатора:

– удаление «пустых» символов (лишние пробелы, переводы строк, табуляции).

– даление комментариев (текст, заключенный в квадратные скобки [...]).

– контроль недопустимых символов.

– распознавание лексем с помощью конечных автоматов (FST).

– формирование таблиц лексем и идентификаторов.

Исходный код программы представлен в приложении А.

3.2 Контроль входных символов

Для проверки входных символов на допустимость используется специальная таблица, которая основана на кодировке Windows-1251. Символы делятся на несколько категорий: T – разрешенные символы (текст), S – пробельные символы и знаки табуляции, F – запрещенные символы, P –

кавычки, 0 – разделители. Реализация контроля входных символов представлена на листинге 3.1.

```
#define                         IN_CODE_TABLE                          {
/*   0   1   2   3   4   5   6   7   8   9   A   B   C   D   E   F*/
/0/ IN::F, IN::F, IN::F, IN::F, IN::F, IN::F, IN::F, IN::F, IN::F,
IN::S,     '|',     IN::F,     IN::F,     IN::F,     IN::F,     IN::F,
/1/ IN::F, IN::F, IN::F, IN::F, IN::F, IN::F, IN::F, IN::F, IN::F,
IN::F,     IN::F,     IN::F,     IN::F,     IN::F,     IN::F,     IN::F,
/2/ IN::S, IN::T, IN::P, 0, IN::F, 0, 0, IN::P, 0, 0, 0, 0, 0, 0,
0,                                                                  0,
/3/ IN::T, IN::T, IN::T, IN::T, IN::T, IN::T, IN::T, IN::T, IN::T,
IN::T,         0,         0,         0,         0,         0,         0,
/4/ 0, IN::T, IN::T, IN::T, IN::T, IN::T, IN::T, IN::T, IN::T,
IN::T,     IN::T,     IN::T,     IN::T,     IN::T,     IN::T,     IN::T,
/5/ IN::T, IN::T, IN::T, IN::T, IN::T, IN::T, IN::T, IN::T, IN::T,
IN::T,       IN::T,       0,       0,       0,       0,       IN::T,
/6/ 0, IN::T, IN::T, IN::T, IN::T, IN::T, IN::T, IN::T, IN::T,
IN::T,     IN::T,     IN::T,     IN::T,     IN::T,     IN::T,     IN::T,
/7/ IN::T, IN::T, IN::T, IN::T, IN::T, IN::T, IN::T, IN::T, IN::T,
IN::T,       IN::T,       0,       0,       0,       0,       IN::F,
/8/ IN::T, IN::T, IN::T, IN::T, IN::T, IN::T, IN::T, IN::T, IN::T,
IN::T,     IN::T,     IN::T,     IN::T,     IN::T,     IN::T,     IN::T,
/9/ IN::T, IN::T, IN::T, IN::T, IN::T, IN::T, IN::T, IN::T, IN::T,
IN::T,     IN::T,     IN::T,     IN::T,     IN::T,     IN::T,     IN::T,
/A/ IN::T, IN::T, IN::T, IN::T, IN::T, IN::T, IN::T, IN::T, IN::T,
IN::T,     IN::T,     IN::T,     IN::T,     IN::T,     IN::T,     IN::T,
/B/ IN::T, IN::T, IN::T, IN::T, IN::T, IN::T, IN::T, IN::T, IN::T,
IN::T,     IN::T,     IN::T,     IN::T,     IN::T,     IN::T,     IN::T,
/C/ IN::T, IN::T, IN::T, IN::T, IN::T, IN::T, IN::T, IN::T, IN::T,
IN::T,     IN::T,     IN::T,     IN::T,     IN::T,     IN::T,     IN::T,
/D/ IN::T, IN::T, IN::T, IN::T, IN::T, IN::T, IN::T, IN::T, IN::T,
IN::T,     IN::T,     IN::T,     IN::T,     IN::T,     IN::T,     IN::T,
/E/ IN::T, IN::T, IN::T, IN::T, IN::T, IN::T, IN::T, IN::T, IN::T,
IN::T,     IN::T,     IN::T,     IN::T,     IN::T,     IN::T,     IN::T,
/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
}
```

Листинг 3.1 – Таблица проверки входных символов

Таблица наглядно показывает допустимость латинских символов и кириллицы в языке ZMI-2025.

3.3 Удаление избыточных символов

Избыточный символ – это символ, удаление которого никак не влияет на исходный текст программы. В языке ZMI-2025 избыточными символами являются комментарии и лишние пробельные символы.

Алгоритм удаления избыточных символов:

– читается символ;

–если встретился символ начала комментария '[' – включается режим игнорирования символов, пока не встретится символ конца комментария ']';

–если символ является запрещенным (категория F) – генерируется ошибка;

–если символ находится внутри строкового литерала (внутри кавычек) – он сохраняется без изменений;

–если символ является пробелом или знаком табуляции (категория S) и не находится внутри строки. Он игнорируется, если предыдущий записанный символ уже был пробелом (сжатие пробелов). Иначе символ записывается в буфер для дальнейшего анализа;

Удаление происходит на этапе ввода исходного текста, перед разбиением входных данных на лексемы.

3.4 Перечень ключевых слов

Лексический анализатор преобразует исходный текст, заменяя лексические единицы лексемами для создания промежуточного представления исходной программы. Соответствие токенов и лексем приведено в таблице 3.1.

Таблица 3.1 – Соответствие токенов и сепараторов с лексемами

Токен	Лексема	Пояснение
byte, text, char, bool	t	Названия типов данных языка.
Идентификатор	i	Имя переменной или функции (макс. 32 символа).
Литерал	l	Литерал любого доступного типа.
proc	f	Объявление функции.
ret	r	Выход из функции/процедуры.
box	m	Точка входа (главная функция).
show	o	Оператор вывода.
check	h	Начало условного оператора (выбор).
is	a	Ветка выбора (case).
else	d	Ветка по умолчанию (default).
;	;	Разделение выражений
,	,	Разделение параметров функций
{	{	Начало блока/тела функции
}	}	Закрытие блока/тела функции
(	(	Передача параметров, приоритет операций.
)	)	Закрытие скобки параметров/приоритета.
=	=	Оператор присваивания
+ - * / %	+ - * / %	Арифметические операторы
== != > < >= <=	v	Операторы сравнения

Пример реализации таблицы лексем представлен в приложении Б.

Каждому выражению соответствует детерминированный конечный автомат, по которому происходит разбор данного выражения.

3.5. Основные структуры данных

Основными структурами данных лексического анализатора являются таблица лексем и таблица идентификаторов. Структура таблицы идентификаторов представлена в листинге 3.1.

```
struct Entry
{
int idxfirstLE; // Индекс в таблице лексем
char id[ID_MAXSIZE]; // Имя идентификатора
IDDATATYPE iddatatype; // Тип данных (INT, STR, CHR, BOOL)
IDTYPE idtype; // Тип (V, F, P, L)
int parm = 0; // Количество параметров (для функций)
union {
int vint; // Значение для byte и bool
char vchar; // Значение для char
struct {
int len;
char str[TI_STR_MAXSIZE - 1];
} vstr; // Значение для text
} value;
};
```

Листинг 3.1 – Структура таблицы лексем

Структура с именем LexTable представляет собой экземпляр таблицы лексем, в которой maxsize – максимальный размер таблицы, size – текущий размер таблицы, table – указатель на строку таблицы лексем. Данная таблица предназначена для хранения записей обо всех лексемах, которые встречаются в исходном коде программы на языке программирования ZMI-2025.

Структура с именем Entry (в пространстве имен LT) представляет экземпляр строки таблицы лексем, в которой lexema хранит символьное обозначение лексемы, sn – номер строки в исходном коде, idxTI – индекс в таблице идентификаторов.

Перечисление с именем IDDATATYPE используется для задания каждому идентификатору типа данных. Для целочисленного INT, для строкового STR, для символьного CHR, для логического BOOL и VOI для процедур.

Перечисление с именем IDTYPE используется для задания каждому идентификатору его типа. Для переменной V, для функции F, для параметра функции P, для литерала L.

Структура с именем IdTable представляет собой экземпляр таблицы идентификаторов. Данная таблица предназначена для хранения уникальных записей обо всех идентификаторах и литералах.

Структура с именем Entry (в пространстве имен IT) представляет экземпляр строки таблицы идентификаторов. Поле idxfirstLE хранит индекс первого вхождения в таблицу лексем. Поле id хранит имя идентификатора (с учетом префикса области видимости). Поле value – это объединение (union), позволяющее хранить значения разных типов (числа, символы, строки) в одной ячейке памяти.

3.6 Структура и перечень сообщений лексического анализатора

В случае возникновения ошибки на стадии лексического анализа формируется экземпляр ошибки в следующем формате: код ошибки в соответствии с таблицей ошибок, текст сообщения, номер строки в исходном коде, позиция в строке. Обработка ошибок производится через механизм исключений (throw ERROR_THROW_IN), что позволяет мгновенно прервать компиляцию при обнаружении некорректной лексемы.

3.7 Принцип обработки ошибок

Если была обнаружена ошибка при выполнении лексического анализа, формируется сообщение об ошибке, которое выводится на консоль и записывается в файл протокола работы (параметр –log). После обнаружения первой ошибки транслятор немедленно прекращает свою работу (механизм исключений).

Перечень сообщений представлен в таблице 3.2.

Таблица 3.2 – Перечень сообщений ошибок лексического анализатора

Номер ошибки	Текст сообщения
201	Неверное имя идентификатора (не распознано ни одной грамматикой)
202	Превышен допустимый размер идентификатора (более 32 символов)
203	Превышен диапазон числового литерала (выход за пределы -128..127)
204	Использование запрещенного имени (совпадение с командами ассемблера)
205	Запрещенный символ (') внутри строкового литерала
206	Использование кириллицы в именах идентификаторов запрещено
207	Неверный формат символьного литерала (ожидался один символ в ' ')

3.8 Параметры лексического анализатора

На вход лексического анализатора подается структура данных In::IN, сформированная на этапе ввода. Она содержит массив слов (лексем), очищенный от пробельных символов и комментариев, а также карту исходного текста для определения позиций ошибок.

3.9 Алгоритм лексического анализа

Лексический анализ выполняется по следующему алгоритму:

– осуществляется последовательный перебор слов из массива in.word;

– если слово является разделителем строк, инкрементируется счетчик;

– для текущего слова последовательно запускаются конечные автоматы

– если ни один автомат не распознал слово, генерируется ошибка «Неверное имя идентификатора»;

– по завершении разбора проверяется наличие и единственность точки входа box;

После выполнения описанного алгоритма лексического анализа, формируется выходной поток токенов, который передается на следующий этап – синтаксический анализ.

3.10 Контрольный пример

Результат работы лексического анализатора в виде таблиц лексем и идентификаторов представлен в приложении Б.

4. Разработка синтаксического анализатора

4.1 Структура синтаксического анализатора

Синтаксический анализ является второй фазой транслятора и выполняется сразу после завершения фазы лексического анализатора. Он предназначен для распознавания заранее заданных синтаксических правил и проверки структуры программы.

Описание структуры синтаксического анализатора языка представлено на рисунке 4.1.

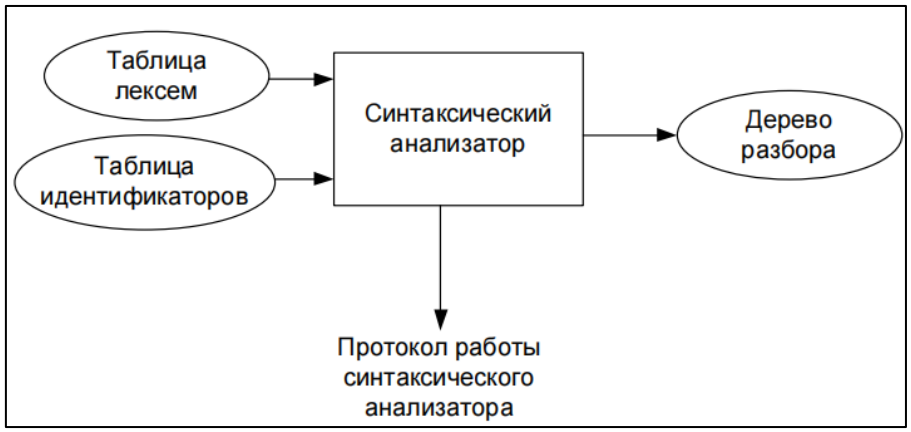

Рисунок 4.1 – Структура синтаксического анализатора

Входными данными для синтаксического анализатора является таблица лексем. Результатом работы является дерево разбора, которое подтверждает правильность синтаксической структуры программы.

4.2 Контекстно-свободная грамматика, описывающая синтаксис языка

В синтаксическом анализаторе транслятора языка ZMI-2025 используется контекстно-свободная грамматика $G = \langle T, N, P, S \rangle$, где

T – множество терминальных символов,

N – множество нетерминальных символов,

P – множество правил языка,

S – начальный символ грамматики.

Данная грамматика G имеет нормальную форму Грейбах, что необходимо для построения нисходящего синтаксического анализатора (автомата с магазинной памятью) и все правила имеют вид A → aα, где a ∈ T, α ∈ N*;S → λ, где S ∈ N – начальный символ, если есть такое правило, то S не должен встречаться в правой части правил.

Все синтаксические правила в форме Грейбах приведены в таблице 4.1.

Таблица 4.1 – Таблица правил переходов нетерминальных символов

Символ	Правила	Какие правила порождает
S	S -> m{N} S -> tf...	Стартовый символ. Описывает структуру программы: либо блок box (m), либо объявление функций (proc).
N	N -> } N- >ti;N N -> i=E;N N -> ovE;N N -> h(E){KN	Основной блок операторов (тело функции или box). Включает: конец блока (}), объявление переменных (byte i;), присваивание (i=...), вывод (show), условный оператор (check).
K	K -> } K -> al:{NK K -> d:{NK	Тело оператора выбора (check). Содержит ветки is (a) и else (d).
E	E -> i E -> l E -> (E)vE E -> i(W)	Выражения. Могут быть переменной (i), литералом (l), сложным выражением в скобках или вызовом функции.
F	F -> ti F -> ti,F	Список формальных параметров при объявлении функции (тип + имя).
W	W -> i W -> l W -> i,W	Список аргументов при вызове функции.

В таблице 4.1 представлено описание нетерминальных символов и соответствующих правил переходов, реализованных в классе Greibach.

4.3 Построение конечного магазинного автомата

Конечный автомат с магазинной памятью (МП-автомат) используется для разбора цепочек, порождаемых контекстно-свободной грамматикой.

Схема работы конечного автомата представлена на рисунке 4.2.

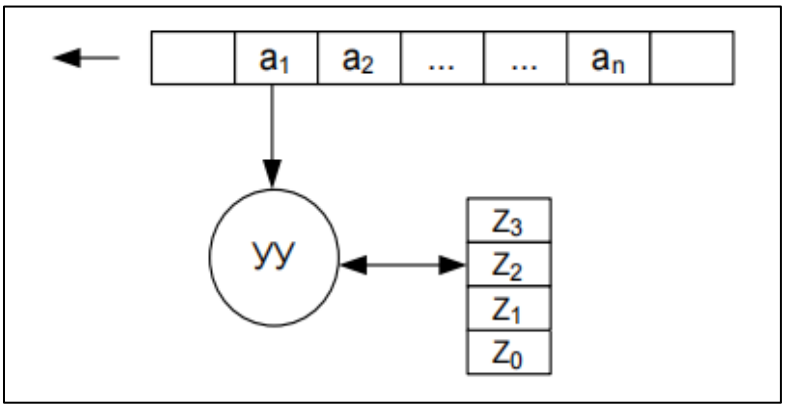

Рисунок 4.2 – Схема работы конечного автомата с магазинной памятью

Алгоритм работы автомата:

– В магазин помещается начальный символ грамматики и маркер дна.

– На каждом шаге автомат считывает очередной символ входной цепочки и сравнивает его с символом, который может быть порожден нетерминалом на вершине магазина.

– Если для нетерминала на вершине магазина существует правило, начинающееся с текущего входного символа, то:

1) Нетерминал удаляется из магазина.

2) В магазин помещается цепочка из правой части правила.

3) Входная лента сдвигается на один символ.

– Если на вершине магазина терминальный символ, и он совпадает с текущим входным — оба удаляются.

– Если магазин пуст и входная цепочка прочитана полностью — разбор успешен. Иначе — ошибка.

Таким образом, в результате работы автомата формируется дерево разбора исходного примера с соблюдением правил языка.

4.4 Основные структуры данных

Основные структуры данных, используемые синтаксическим анализатором, представлены в таблице 4.2.

Таблица 4.2 – Структуры синтаксического анализатора

Структура	Характеристики
1	2
Greibach	Структура для представления контекстно-свободной грамматики в памяти. Все символы (алфавит) представляются в формате GRBALPHABET (short). Терминалы — положительные значения, нетерминалы — отрицательные. Поля: – rules: массив правил (типа Rule); – size: количество правил; – startN: стартовый символ грамматики; – stbottomT: символ дна стека. Методы: – getRule: поиск правила по нетерминалу или номеру
Rule	Структура, представляющая одно правило грамматики вида `A ->α1
Rule::Chain	Структура, представляющая одну цепочку символов (правую часть правила). Поля: – size: длина цепочки; – nt: массив символов цепочки. Методы: – getCChain: возвращает строковое представление цепочки; – isT / isN: проверка, является ли символ терминалом или нетерминалом – alphabet_to_char: преобразование внутреннего кода символа в char.

Продолжение таблицы 4.2

1	2
MfstState	Структура для сохранения состояния автомата (используется для возврата назад / backtracking). Поля: – lenta_position: текущая позиция на входной ленте; – nrule: индекс текущего правила; – nrulechain: индекс текущей цепочки; – st: копия стека автомата на момент сохранения.
Mfst	Основная структура, реализующая магазинный автомат (МП-автомат). Поля: – lenta: массив входных лексем (преобразованных в GRBALPHABET); – lenta_position: текущая позиция считывающей головки; – st: стек автомата; – storestate: стек сохраненных состояний (для возврата при неудаче); – greibach: используемая грамматика; – lex: таблица лексем; – log: журнал для записи протокола; – diagnosis: массив структур для диагностики ошибок. Методы: – start: запуск синтаксического анализа; – step: выполнение одного шага автомата; – push_chain: помещение цепочки в стек; – savestate / resetstate: сохранение и восстановление состояния; – savediagnosis: сохранение информации об ошибке для последующего вывода
MfstDiagnosis	Вспомогательная структура для хранения информации о последней ошибке (используется для формирования понятных сообщений пользователю). Поля: – lenta_position: позиция ошибки; – rc_step: код возврата шага; – nrule: ожидаемое правило; – nrule_chain: ожидаемая цепочка.

С помощью представленных конструкций формируется логика создания дерева разбора и цепочки правил.

4.5 Описание алгоритма синтаксического разбора

Принцип работы автомата:

– в магазин записывается стартовый символ;

– на основе полученных ранее таблиц формируется входная лента;

– запускается автомат;

– выбирается цепочка, соответствующая нетерминальному символу, записывается в магазин в обратном порядке;

– если терминалы в стеке и в ленте совпадают, то данный терминал удаляется из ленты и стека. Иначе возвращаемся в предыдущее сохраненное состояние и выбираем другую цепочку нетерминала;

– если в магазине встретился нетерминал, переходим к пункту 4;

– если наш символ достиг дна стека, и лента в этот момент пуста, то синтаксический анализ выполнен успешно. Иначе генерируется исключение.

4.6 Параметры синтаксического анализатора

Входной информацией для синтаксического анализатора являются таблица лексем (поток токенов) и грамматика языка в нормальной форме Грейбах (структура Greibach). Результаты работы – дерево разбора (последовательность примененных правил) и протокол работы автомата – выводятся в файл журнала (.log) и частично в консоль для отладки.

4.7 Структура и перечень сообщений синтаксического анализатора

При возникновении ошибки синтаксический анализатор формирует сообщение, содержащее код ошибки, поясняющий текст, номер строки и символ, на котором произошел сбой.

Все сообщения синтаксического анализатора с кодом ошибки представлены в таблице 4.3.

Таблица 4.3 – Сообщения синтаксического анализатора с кодом ошибки

Номер ошибки	Текст сообщение
600	Неверная структура программы (ожидался box, proc или })
601	Неверный оператор (лишняя ';' или неизвестная команда)
602	Ошибка в выражении (ожидался операнд или скобка)
603	Ошибка в параметрах объявления функции
604	Ошибка в аргументах вызова функции
606	Неверная структура условия (в конструкции check)
607	Неожиданный конец файла (возможно, пропущена '}')

При выводе кода ошибки также уточняется на каком этапе анализа произошла ошибка.

4.8 Принцип обработки ошибок

Механизм обработки ошибок базируется на диагностике состояния автомата в тупиковых ситуациях:

Если автомат не может подобрать правило для текущего нетерминала или терминалы не совпадают, он сохраняет информацию о текущей позиции и ожидаемом правиле в структуру диагностики (diagnosis). После этого выполняется попытка возврата к предыдущему состоянию (backtracking).

Если все варианты перебраны и разбор невозможен, транслятор останавливает работу. Пользователю выдается сообщение об ошибке,

соответствующее той попытке разбора, которая продвинулась максимально далеко по тексту программы (наиболее вероятное место ошибки).

4.9 Контрольный пример

Результаты работы синтаксического анализатора, включая построенное дерево разбора и протокол состояний автомата для контрольного примера, приведены в приложении В.

5 Разработка семантического анализатора

5.1 Структура семантического анализатора

Семантический анализатор предназначен для проверки смысловой корректности программы. Он выявляет ошибки, которые невозможно обнаружить на этапе синтаксического анализа: несоответствие типов данных, нарушение правил области видимости, некорректное использование функций и операторов.

Общая структура семантического анализатора представлена на рисунке 5.1.

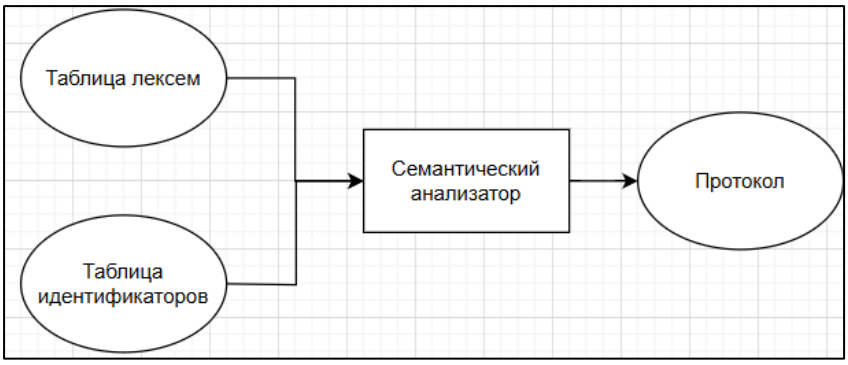

Рисунок 5.1 – Структура семантического анализатора

Входными данными для семантического анализатора являются таблица лексем и таблица идентификаторов, сформированные на предыдущих этапах. Выходным результатом является протокол работы и список обнаруженных ошибок на всех этапах анализа.

5.2 Функции семантического анализатора

Семантический анализатор выполняет проверку правил, описанных в разделе 1.16, включая контроль типов при присваивании и арифметических операциях, проверку параметров функций и корректность структуры оператора выбора.

5.3 Структура и перечень сообщений семантического анализатора

Перечень сообщений, формируемых семантическим анализатором, представлен в таблице 5.1.

Таблица 5.1 – Перечень сообщений семантического анализатора

Номер ошибки	Текст сообщение
1	2
301	Отсутствует точка входа box
302	Обнаружено несколько точек входа box
304	Использование необъявленного идентификатора
305	Попытка переопределения идентификатор
308	Кол-во ожидаемых и переданных параметров не совпадает
309	Несовпадение типов передаваемых параметров

Продолжение таблицы 5.1

1	2
314	Типы данных в выражении не совпадают (попытка смешивания text и byte)
315	Тип функции и возвращаемого значения не совпадают
318	Попытка деления на ноль
319	Оператор check обязан содержать ветку else
320	Дублирование значения is (case)
321	Множественное использование ветки else (default)

При выводе кода ошибки также уточняется на каком этапе анализа произошла ошибка.

5.4 Принцип обработки ошибок

При обнаружении ошибки в исходном коде программы семантический анализатор формирует сообщение с указанием кода, текста ошибки и места её возникновения (номер строки). Сообщение выводится в консоль и записывается в файл протокола (.log). Анализатор продолжает проверку кода до конца, чтобы выявить максимально возможное количество ошибок за один проход, однако генерация объектного кода в случае наличия семантических ошибок блокируется.

5.5 Контрольный пример

Примеры диагностики семантических ошибок приведены в таблице 5.2.

Таблица 5.2. – Примеры диагностики ошибок

Пример кода	Ошибка

box {
byte x;
x = 9;
text y;
y = x;
}
	Ошибка №314: Семантическая ошибка: Типы данных в выражении не совпадают.
	Строка: 5

box {
byte x;
}
box {
text y;
}
	Ошибка №302: Семантическая ошибка: Обнаружено несколько точек входа box.
	Строка: 0

check(val) {
is 0: { ... }
}
	Ошибка №319: Семантическая ошибка: Оператор check обязан содержать ветку else.
	Строка: (номер строки закрывающей скобки)

Ошибки записываются в протокол с указанием номера ошибки и диагностического сообщения.

6. Вычисление выражений

6.1 Выражения, допускаемые языком

В языке ZMI-2025 допускаются вычисления выражений целочисленного типа данных с поддержкой вызова функций внутри выражений. Приоритет операций представлен на таблице 6.1.

Таблица 6.1 – Приоритеты операций

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

Операторы в скобках имеют наивысший приоритет, за ними следуют умножение и деление, а затем сложение и вычитание.

6.2 Польская запись и принцип её построения

Польская запись является альтернативным способом записи арифметических выражений, которые затем удобно вычислять с помощью стека. Обратная польская запись – это форма записи выражений, когда операнды идут перед знаками операций.

Алгоритм построения обратной польской записи:

– если встретился операнд (идентификатор или литерал), то он переносится в строку результата;

–если встретилась открывающаяся скобка, то она помещается в стек;

– если встретилась закрывающаяся скобка, то извлекаются все операции из стека в результирующую строку, пока не встретится открывающаяся скобка, которая затем удаляется из стека;

– если встретился вызов функции, то её аргументы обрабатываются как обычные выражения, а после последнего аргумента в результирующую строку добавляется специальный оператор вызова «@»;

– если встретилась операция, то сравнивается её приоритет с операцией на вершине стека. Если приоритет текущей операции меньше или равен приоритету операции в стеке, то операции из стека извлекаются в результат. После этого текущая операция помещается в стек;

– если всё выражение просмотрено, то все оставшиеся в стеке операции переносятся в результирующую строку.

После создания польской записи, а именно перезаписи таблицы лексем в постфиксную форму записи, измененный строки с лексемами будут

переданы в код генерации ассемблера.

6.3 Программная реализация обработки выражений

Программная реализация алгоритма преобразования выражений к польской записи выполнена в модуле PN.cpp и представлена в приложении Г.

6.4 Контрольный пример

Пример преобразования выражений из актуального контрольного примера к обратной польской записи представлен в таблице 6.2.

Таблица 6.2. – Преобразование выражений к ПОЛИЗ

Выражение	Обратная польская запись для выражения
res = (a + b) * 2;	res a b + 2 * =
res = 100 / 3;	res 100 3 / =
res = calc(10, 5);	res 10 5 @ =
res = mabs(-50);	res -50 @ =
lenVal = strle("Привет");	lenVal "Привет" @ =

Преобразование в формат ПОЛИЗ необходимо для реализации стековой модели вычислений на языке ассемблера. В приложении Г приведены результаты декодирования польской записи, отображающие финальный порядок операндов и операторов.

7. Генерация кода

7.1 Структура генератора кода

Трансляция языка ZMI-2025 производится в язык ассемблера. Структура генератора кода ZMI-2025 представлена на рисунке 7.1.

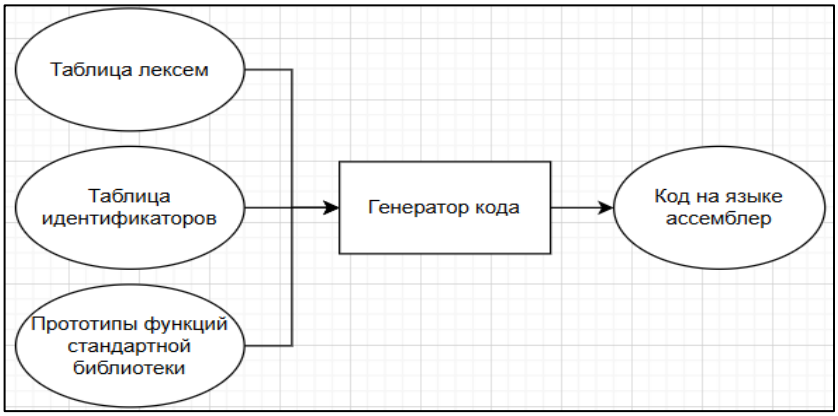

Рисунок 7.1 – Структура генератора кода

Генерация базируется на последовательном просмотре таблицы лексем, которая предварительно была обработана и приведена к Обратной Польской Записи (ПОЛИЗ). Функции исходного языка транслируются в процедуры (PROC) на ассемблере. Вычисления выражений реализуются через стековую модель процессора x86 (команды push, pop, add, imul). Управляющие конструкции (check) преобразуются в набор меток и команд условного перехода (cmp, jne, jmp).

7.2 Представление типов данных в оперативной памяти

Соответствия между типами данных на языке ZMI-2025 и на языке ассемблера приведены в таблице 7.1.

Таблица 7.1 – Соответствия типов идентификаторов языка и ассемблера

Тип идентификатора на языке ZMI-2025	Тип идентификатора на языке ассемблера	Пояснение
byte, bool, char	sbyte	наковое 8-битное целое число (1 байт). Хранит числа от -128 до 127, коды символов или логические 0/1.
text	dd	32-битный указатель (адрес). Сама строка хранится в сегменте констант как массив байтов, завершающийся нулем (db '...', 0). Переменная типа text хранит адрес начала этой строки.

Элементы таблицы идентификаторов размещаются в сегментах .data (переменные) и .const (литералы).

7.3 Статическая библиотека

В языке ZMI-2025 предусмотрена статическая библиотека InLib.lib, написанная на языке C++. Объявление функций библиотеки генерируется в коде ассемблера автоматически с использованием директивы PROTO.

Таблица 7.3 – Функции статической библиотеки

Функция	Назначение
outstr PROTO :DWORD	Вывод строки в стандартный поток (с поддержкой кириллицы).
outnum PROTO :SDWORD	Вывод знакового целого числа.
newline PROTO	Переход на новую строку.
strle PROTO :DWORD	Вычисление длины строки.
stcmp PROTO :DWORD, :DWORD	Лексикографическое сравнение двух строк.
strtoint PROTO :DWORD	Преобразование строки в число.
mabs PROTO :SDWORD	Вычисление модуля числа.
rnd PROTO :SDWORD	Генерация случайного числа.

На этапе генерации кода библиотека подключается директивой includelib.

7.4 Особенности алгоритма генерации кода

Генерация кода выполняется в один проход по таблице лексем. Общая схема работы генератора кода представлена на рисунке 7.2.

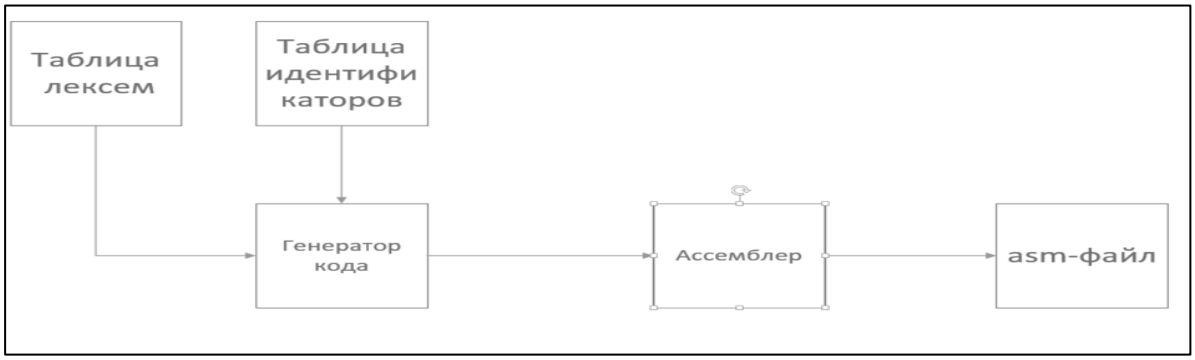

Рисунок 7.2 – Структура генератора кода

Так как выражения представлены в ПОЛИЗе, генератор использует стек процессора. Операнды (переменные, литералы) помещаются в стек командой push. Операторы (+, -, *) извлекают значения из стека (pop), производят вычисление и помещают результат обратно. Кроме того, оператор show < анализирует тип предыдущего операнда. Если это строка, генерируется вызов outstr, если число – outnum. Также

конструкция check реализуется через стек состояний (для поддержки вложенности). Для каждой ветки is генерируется сравнение и условный переход к следующей ветке или выходу.

7.5 Параметры, управляющие генерацией кода

На вход генератору кода поступают таблицы лексем (сформированные лексером и модифицированные модулем ПОЛИЗ) и таблица идентификаторов. Результатом работы является текстовый файл с расширением .asm, готовый для трансляции ассемблером ML.EXE.

7.6 Контрольный пример

Результат генерации ассемблерного кода на основе контрольного примера из приложения А приведен в приложении Д.

8. Тестирование транслятора

8.1 Общие положения

Тесты используются для проверки транслятора на работоспособность и для выявления недочетов и ошибок, а также последующего их исправления. При возникновении ошибки транслятор завершает свою работу и выводит сообщение об ошибке с соответствующим ей кодом на консоль и в файл протокола .log

8.2 Результаты тестирования

В языке ZMI-2025 не разрешается использовать запрещённые входным алфавитом символы. Результат использования запрещённого символа показан в таблице 8.1..

Таблица 8.1 – Тестирование проверки на допустимость символов

Исходный код	Диагностическое сообщение

box {
$
}
	Ошибка 111: Недопустимый символ в исходном файле (-in)
	Строка: 2 Позиция: 2

На этапе лексического анализа в языке ZMI-2025 могут возникнуть ошибки, связанные с неверными идентификаторами или литералами. Результаты тестирования лексического анализатора показаны в таблице 8.2.

Таблица 8.2 - Тестирование лексического анализатора

Исходный код	Диагностическое сообщение

box {
byte 1num;
}
	Ошибка 201: Лексический анализатор: Неверное имя идентификатора (начинается с цифры)
	Строка: 2 Позиция: 74

box {
byte число;
}
	Ошибка 206: Лексический анализатор: Использование кириллицы в идентификаторах запрещено
	Строка: 2 Позиция: 7

box {
text s;
s = "Don't";
}
	Ошибка 205: Лексический анализатор: Запрещенный символ (') в строке
	Строка: 3 Позиция: 11

На этапе синтаксического анализа проверяется структура программы. Результаты тестирования синтаксического анализатора показаны в таблице 8.3.

Таблица 8.3 – Тестирование синтаксического анализатора

Исходный код	Диагностическое сообщение
1	2
box { byte x }	Ошибка 601: Синтаксическая ошибка: Неверный оператор (лишняя ';' или неизвестная команда).

Продолжение таблицы 8.3

1	2
text proc fi({}	Ошибка 603: Синтаксическая ошибка: Ошибка в параметрах функции

box {
byte x;
x = ;
}
	Ошибка 602: Синтаксическая ошибка: Ошибка в выражении

box { byte x;	Ошибка 607: Синтаксическая ошибка: Неожиданный конец файла (возможно, пропущена '}')

Семантический анализ в языке ZMI-2025 содержит множество проверок по правилам типов и логики. Итоги тестирования семантического анализатора приведены в таблице 8.4.

Таблица 8.4 – Тестирование семантического анализатора

Исходный код	Диагностическое сообщение
1	2
byte proc f() { ret 0; }	Ошибка 301: Семантическая ошибка: Отсутствует точка входа main
box {} box {}	Ошибка 302: Семантическая ошибка: Обнаружено несколько точек входа main

box {
byte a; byte a;
}
	Ошибка 305: Семантическая ошибка: Попытка переопределения идентификатора строка 4 позиция 0

byte proc f(byte x) { ret 0; } box { f(); }	Ошибка 308: Семантическая ошибка: Кол-во ожидаемых и переданных параметров не совпадает

byte proc f(byte x) { ret 0; }
box {
text s = "1";
f(s);
}
	Ошибка 309: Семантическая ошибка: Несовпадение типов передаваемых параметров

box {
text s = "";
}
	Ошибка 310: Семантическая ошибка: Использование пустого строкового литерала недопустимо

box {
byte x;
x = "text";
}
	Ошибка 314: Семантическая ошибка: Типы данных в выражении не совпадают

box {
byte x;
x = 100 / 0;
}
	Ошибка 318: Семантическая ошибка: Попытка деления на ноль

box {
check(1) {
is 0: { }
}
}
	Ошибка 319: Семантическая ошибка: Оператор check обязан содержать ветку else

box { check(1) {	Ошибка 320: Семантическая ошибка: Дублирование значения is (case)

Продолжение таблицы 8.4

1	2
is 0: {} is 0: {} else: {} } }	

Ошибка семантического анализатора также приводит к прекращению выполнения программы и записи соответствующей ошибки в лог журнал.

Заключение

В ходе выполнения курсовой работы был разработан компилятор для языка программирования ZMI-2025, осуществляющий трансляцию исходного кода в язык ассемблера MASM.

В процессе разработки были успешно решены следующие задачи:

– Сформулирована спецификация языка ZMI-2025, включая уникальный синтаксис (box, check, show) и систему типов.

– Разработан лексический анализатор на основе конечных автоматов (FST), поддерживающий работу с кириллицей и различными системами счисления.

– Построена контекстно-свободная грамматика в нормальной форме Грейбах и реализован синтаксический анализатор на основе автомата с магазинной памятью.

– Реализован семантический анализатор, выполняющий контроль типов, областей видимости и логической целостности программы.

– Разработан алгоритм преобразования выражений в Обратную Польскую Запись (ПОЛИЗ) для реализации стековой модели вычислений.

– Создан генератор объектного кода, формирующий валидный файл формата .asm, готовый к компиляции и компоновке с системными библиотеками.

– Проведено комплексное тестирование всех модулей транслятора с использованием контрольных примеров.

Окончательная версия языка ZMI-2025 обладает следующими характеристиками:

– 4 типа данных: целочисленный (byte), строковый (text), символьный (char) и логический (bool).

– Универсальный оператор вывода show <, поддерживающий различные типы данных.

–Управляющие конструкции: точка входа box, оператор множественного выбора check с поддержкой вложенности.

–Богатая стандартная библиотека: функции работы со строками, математические функции и генерация случайных чисел.

–Поддержка кириллицы в строковых литералах и комментариях.

– Развитая система диагностики ошибок на всех этапах трансляции.

Проделанная работа позволила закрепить теоретические знания в области системного программирования и получить практические навыки разработки сложных программных систем, включающих этапы лексического, синтаксического и семантического анализа, а также генерации машинозависимого кода.

Список использованных источников

1 Habr – Об изучении компиляторов и создании языков программирования [Электронный ресурс]. – Режим доступа: https://habr.com/ru/search/.

2 Построение компиляторов / Никлаус Вирт 2010. – 194 с.

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

4 Язык программирования C++.Лекции и упражнения [6-е издание] / Стивен Прата 2019 – 1094 с.

Приложение А

Контрольный пример

```
byte proc strtoint(text strArg) { ret 0; }
byte proc stcmp(text sOne, text sTwo) { ret 0; }
byte proc strle(text txtArg) { ret 0; }
byte proc mabs(byte nArg) { ret 0; }
byte proc rnd(byte maxArg) { ret 0; }
[ Функция ]
byte proc calc(byte a, byte b) {
    byte res;
    res = (a + b) * 2;
    ret res;
}
box {
    text welcome;
    byte numDec;
    byte numHex;
    byte numBin;
    char myChar;
    byte myBool;
    byte res;
    byte lenVal;
    byte rndVal;
    welcome = "Типы данных";
    welcome = "Строковый тип данных";
    show < welcome;
    numDec = 100;
    show < "Decimal (100):";
    show < numDec;
    numHex = 0xFF;
    show < "Hex (0xFF -> -1):";
    show < numHex;
    numBin = 0b101;
    show < "Binary (0b101):";
    show < numBin;
    myChar = 'Z';
    show < "Char (Z -> 90):";
    show < myChar;
    [ Логический: true=1, false=0 ]
    myBool = true;
    show < "Bool (true -> 1):";
    show < myBool;
    [ Арефмитические операции ]
    show < "Math | Математика";
    res = 100 / 3;
    show < "100 / 3 =";
    show < res;
    res = 100 % 3;
    show < "100 % 3 =";
    show < res;
```

```
    res = calc(10, 5);
    show < "calc(10, 5) -> (10+5) * 2 =";
    show < res;
    res = mabs(-50);
    show < "mabs(-50) =";
    show < res;
    [ Операции со строками ]
    show < "Strings | Строки ";
    lenVal = strle("Привет");
    show < "Length of Privet (expect 6):";
    show < lenVal;
    res = strtoint("42");
    show < "String 42 to Byte:";
    show < res;
    [ Операции сравнения ]
    show < "Comparison | Сравнение ";
    res = 10 == 10;
    show < "10 == 10 (Expect 1):";
    show < res;
    res = 10 != 5;
    show < "10 != 5 (Expect 1):";
    show < res;
    res = 20 > 10;
    show < "20 > 10 (Expect 1):";
    show < res;
    res = 5 < 10;
    show < "5 < 10 (Expect 1):";
    show < res;
    res = 10 <= 10;
    show < "10 <= 10 (Expect 1):";
    show < res;
    res = 5 >= 10;
    show < "5 >= 10 (Expect 0):";
    show < res;
    [ Условный оператор ]
    show < "Conditional operator | Условный оператор";
    rndVal = rnd(3);
    show < "Random (0 - 2):";
    show < rndVal;
    check(rndVal) {
        is 0: {
            show < "Case 0: Zero/Ноль";
        }
        is 1: {
            show < "Case 1: One/Один";
        }
        else: {
            show < "Default/По умолчанию";
        }
    }
    show < "The end | Конец";}
```

Листинг 1 – Контрольный пример языка

Приложение Б

```
------------------Таблица индетификаторов------------------

 Идентификатор   | тип данных | первое вхождение | содержание
-----------------|------------|------------------|------------
_____Литерал_____________________________________________
L1               | int(1)     | 0                | 0
L2               | int(1)     | 0                | 2
L3               | string     | 0                | [11]Типы
данных
L4               | string     | 0                | [20]Строковый
тип данных
L5               | int(1)     | 0                | 100
L6               | string     | 0                | [14]Decimal
(100):
L7               | int(1)     | 0                | 255
L8               | string     | 0                | [17]Hex (0xFF
-> -1):
L9               | int(1)     | 0                | 5
L10              | string     | 0                | [15]Binary
(0b101):
L11              | char       | 0                | 'Z'
L12              | string     | 0                | [15]Char (Z ->
90):
true_lit         | int(1)     | 0                | 1
L13              | string     | 0                | [17]Bool (true
-> 1):
L14              | string     | 0                | [17]Math |
Математика
L15              | int(1)     | 0                | 3
L16              | string     | 0                | [9]100 / 3 =
L17              | string     | 0                | [9]100 % 3 =
L18              | int(1)     | 0                | 10
L19              | int(1)     | 0                | 5
L20              | string     | 0                | calc(10, 5) ->
(10+5) * 2...
L21              | int(1)     | 0                | -50
L22              | string     | 0                | [11]mabs(-50)
=
L23              | string     | 0                | [17]Strings |
Строки
L24              | string     | 0                | [6]Привет
L25              | string     | 0                | Length of
Privet (expect ...
L26              | string     | 0                | [2]42
L27              | string     | 0                | [18]String 42
to Byte:
L28              | string     | 0                | [23]Comparison
| Сравнение
L29              | string     | 0                | [20]10 == 10
(Expect 1):
```

```
L30             | string     | 0                  | [19]10 != 5
(Expect 1):
L31             | int(1)     | 0                  | 20
L32             | string     | 0                  | [19]20 > 10
(Expect 1):
L33             | string     | 0                  | [18]5 < 10
(Expect 1):
L34             | string     | 0                  | [20]10 <= 10
(Expect 1):
L35             | string     | 0                  | [19]5 >= 10
(Expect 0):
L36             | string     | 0                  | Conditional
operator | Ус...
L37             | string     | 0                  | [15]Random (0
- 2):
L38             | string     | 0                  | [17]Case 0:
Zero/Ноль
L39             | string     | 0                  | [16]Case 1:
One/Один
L40             | string     | 0                  | [20]Default/По
умолчанию
L41             | string     | 0                  | [15]The end |
Конец
_____Переменная_____________________________________________
calcres         | int(1)     | 75                 | -
welcome         | string     | 94                 | -
numDec          | int(1)     | 97                 | -
numHex          | int(1)     | 100                | -
numBin          | int(1)     | 103                | -
myChar          | char       | 106                | -
myBool          | int(1)     | 109                | -
res             | int(1)     | 112                | -
lenVal          | int(1)     | 115                | -
rndVal          | int(1)     | 118                | -
_____Параметр_______________________________________________
strArg          | string     | 5                  | -
sOne            | string     | 17                 | -
sTwo            | string     | 20                 | -
txtArg          | string     | 32                 | -
nArg            | int(1)     | 44                 | -
maxArg          | int(1)     | 56                 | -
a               | int(1)     | 68                 | -
b               | int(1)     | 71                 | -
_____Функция________________________________________________
strtoint        | int(1)     | 2                  | -
stcmp           | int(1)     | 14                 | -
strle           | int(1)     | 29                 | -
mabs            | int(1)     | 41                 | -
rnd             | int(1)     | 53                 | -
calc            | int(1)     | 65                 | -
```

Листинг 1 – Таблица идентификаторов контрольного примера

Приложение В

```
namespace GRB {
    char* Rule::Chain::getCChain(char* b) {
        for (int i = 0; i < size; i++) b[i] =
Chain::alphabet_to_char(nt[i]);
        b[size] = 0; return b;
    }

    void Rule::AddChain(const std::vector<GRBALPHABET>& chain) {
        if (size >= GRB_MAX_CHAINS) return;
        chains[size].size = (short)chain.size();
        for (size_t i = 0; i < chain.size(); i++) {
            if (i < GRB_MAX_LEN) chains[size].nt[i] = chain[i];
        }
        size++;
    }

    char* Rule::getCRule(char* b, short nchain) {
        char bchain[200];
        b[0] = Chain::alphabet_to_char(nn);
        b[1] = '-'; b[2] = '>'; b[3] = 0;
        chains[nchain].getCChain(bchain);
        strcat_s(b, 200, bchain);
        return b;
    }

    short Rule::getNextChain(GRBALPHABET t, Rule::Chain& pchain,
short j) {
        short rc = -1;
        while (j < size && chains[j].nt[0] != t) j++;
        rc = (j < size ? j : -1);
        if (rc >= 0) pchain = chains[rc];
        return rc;
    }

    short Greibach::getRule(GRBALPHABET pnn, Rule& prule) {
        short rc = -1;
        short k = 0;
        while (k < size && rules[k].nn != pnn) k++;
        if (k < size) { prule = rules[k]; rc = k; }
        return rc;
    }

    Rule Greibach::getRule(short n) { return n < size ? rules[n]
: Rule(); }

    Greibach getGreibach()
    {
        Greibach g;
        g.startN = NS('S');
        g.stbottomT = TS('$');
        g.size = 8;
```

```
        g.rules[0].nn = NS('S'); g.rules[0].iderror =
GRB_ERROR_SERIES + 0;

        g.rules[0].AddChain({ TS('m'), TS('{'), NS('N') });

        g.rules[0].AddChain({ TS('t'), TS('f'), TS('i'),
TS('('), NS('F'), TS(')'), TS('{'), NS('N'), NS('S') });
        g.rules[0].AddChain({ TS('t'), TS('f'), TS('i'),
TS('('), TS(')'),        TS('{'), NS('N'), NS('S') });
        g.rules[0].AddChain({ TS('s'), TS('f'), TS('i'),
TS('('), NS('F'), TS(')'), TS('{'), NS('N'), NS('S') });
        g.rules[0].AddChain({ TS('s'), TS('f'), TS('i'),
TS('('), TS(')'),        TS('{'), NS('N'), NS('S') });
        g.rules[0].AddChain({ TS('c'), TS('f'), TS('i'),
TS('('), NS('F'), TS(')'), TS('{'), NS('N'), NS('S') });
        g.rules[0].AddChain({ TS('c'), TS('f'), TS('i'),
TS('('), TS(')'),        TS('{'), NS('N'), NS('S') });
        g.rules[0].AddChain({ TS('n'), TS('f'), TS('i'),
TS('('), NS('F'), TS(')'), TS('{'), NS('N'), NS('S') });
        g.rules[0].AddChain({ TS('n'), TS('f'), TS('i'),
TS('('), TS(')'),        TS('{'), NS('N'), NS('S') });

        g.rules[1].nn = NS('N'); g.rules[1].iderror =
GRB_ERROR_SERIES + 1;

        g.rules[1].AddChain({ TS('}') });

        g.rules[1].AddChain({ TS('t'), TS('i'), TS(';'), NS('N')
});
        g.rules[1].AddChain({ TS('s'), TS('i'), TS(';'), NS('N')
});
        g.rules[1].AddChain({ TS('c'), TS('i'), TS(';'), NS('N')
});

        g.rules[1].AddChain({ TS('i'), TS('='), NS('E'),
TS(';'), NS('N') });

        g.rules[1].AddChain({ TS('i'), TS('('), NS('W'),
TS(')'), TS(';'), NS('N') });
        g.rules[1].AddChain({ TS('i'), TS('('), TS(')'),
TS(';'), NS('N') });

        g.rules[1].AddChain({ TS('o'), TS('v'), NS('E'),
TS(';'), NS('N') });

        g.rules[1].AddChain({ TS('r'), NS('E'), TS(';'), NS('N')
});

        g.rules[1].AddChain({ TS('h'), TS('('), NS('E'),
TS(')'), TS('{'), NS('K'), NS('N') });
```

```
        g.rules[2].nn = NS('K'); g.rules[2].iderror =
GRB_ERROR_SERIES + 6;

        g.rules[2].AddChain({ TS('}') });

        g.rules[2].AddChain({ TS('a'), TS('l'), TS(':'),
TS('{'), NS('N'), NS('K') });

        g.rules[2].AddChain({ TS('d'), TS(':'), TS('{'),
NS('N'), NS('K') });

        g.rules[3].nn = NS('E'); g.rules[3].iderror =
GRB_ERROR_SERIES + 2;

        g.rules[3].AddChain({ TS('('), NS('E'), TS(')'),
TS('v'), NS('E') });
        g.rules[3].AddChain({ TS('('), NS('E'), TS(')') });

        g.rules[3].AddChain({ TS('i'), TS('v'), NS('E') });
        g.rules[3].AddChain({ TS('l'), TS('v'), NS('E') });

        g.rules[3].AddChain({ TS('i'), TS('('), NS('W'), TS(')')
});
        g.rules[3].AddChain({ TS('i'), TS('('), TS(')') });

        g.rules[3].AddChain({ TS('i') });
        g.rules[3].AddChain({ TS('l') });

        g.rules[4].nn = NS('F'); g.rules[4].iderror =
GRB_ERROR_SERIES + 3;
        g.rules[4].AddChain({ TS('t'), TS('i'), TS(','), NS('F')
});
        g.rules[4].AddChain({ TS('s'), TS('i'), TS(','), NS('F')
});
        g.rules[4].AddChain({ TS('c'), TS('i'), TS(','), NS('F')
});
        g.rules[4].AddChain({ TS('t'), TS('i') });
        g.rules[4].AddChain({ TS('s'), TS('i') });
        g.rules[4].AddChain({ TS('c'), TS('i') });

        g.rules[5].nn = NS('W'); g.rules[5].iderror =
GRB_ERROR_SERIES + 4;
        g.rules[5].AddChain({ TS('i'), TS(','), NS('W') });
        g.rules[5].AddChain({ TS('l'), TS(','), NS('W') });
        g.rules[5].AddChain({ TS('i') });
        g.rules[5].AddChain({ TS('l') });

        return g;
    }
}
```

Листинг 1 – Грамматика языка ZMI-2025

```
#define GRB_ERROR_SERIES 600
```

```
#define GRB_MAX_CHAINS 32
#define GRB_MAX_LEN 32
#define GRB_MAX_RULES 32
typedef short GRBALPHABET;
#define NS(n) GRB::Rule::Chain::N(n)
#define TS(n) GRB::Rule::Chain::T(n)
#define ISNS(n) GRB::Rule::Chain::isN(n)
namespace GRB{
     struct Rule
     {
          GRBALPHABET nn;
          int iderror;
          short size;

          struct Chain
          {
               short size;
               GRBALPHABET nt[GRB_MAX_LEN];

               Chain() { size = 0; }

               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); }

               char* getCChain(char* b);
          };

          Chain chains[GRB_MAX_CHAINS];

          Rule() { nn = 0x00; size = 0; iderror = -1; }

          void AddChain(const std::vector<GRBALPHABET>& chain);

          char* getCRule(char* b, short nchain);
          short getNextChain(GRBALPHABET t, Rule::Chain& pchain,
short j);
     };

     struct Greibach
     {
          short size;
          GRBALPHABET startN;
          GRBALPHABET stbottomT;

          Rule rules[GRB_MAX_RULES];

          Greibach() { size = 0; startN = 0; stbottomT = 0; }
```

```
        short getRule(GRBALPHABET pnn, Rule& prule);
        Rule getRule(short n);
    };

    Greibach getGreibach();
};
```

Листинг 2 – Структура грамматики Грейбах

```
          Шаг  :Правило                          Входная лента
                                      Стек
0    : S->tfi(F){NS                      tfi(si){rl;}tfi(si,si){rl
S$
0    : SAVESTATE:          1
0    :                                   tfi(si){rl;}tfi(si,si){rl
tfi(F){NS$
1    :                                   fi(si){rl;}tfi(si,si){rl;
fi(F){NS$
2    :                                   i(si){rl;}tfi(si,si){rl;}
i(F){NS$
3    :                                   (si){rl;}tfi(si,si){rl;}t
(F){NS$
4    :                                   si){rl;}tfi(si,si){rl;}tf
F){NS$
5    : F->si,F                           si){rl;}tfi(si,si){rl;}tf
F){NS$
5    : SAVESTATE:          2
5    :                                   si){rl;}tfi(si,si){rl;}tf
si,F){NS$
6    :                                   i){rl;}tfi(si,si){rl;}tfi
i,F){NS$
7    :                                   ){rl;}tfi(si,si){rl;}tfi(
,F){NS$
8    : TS_NOK / NS_NORULECHAIN
8    : RESSTATE
8    :                                   si){rl;}tfi(si,si){rl;}tf
F){NS$
9    : F->si                             si){rl;}tfi(si,si){rl;}tf
F){NS$
9    : SAVESTATE:          2
9    :                                   si){rl;}tfi(si,si){rl;}tf
si){NS$
10   :                                   i){rl;}tfi(si,si){rl;}tfi
i){NS$
11   :                                   ){rl;}tfi(si,si){rl;}tfi(
){NS$
12   :                                   {rl;}tfi(si,si){rl;}tfi(s
{NS$
13   :                                   rl;}tfi(si,si){rl;}tfi(si
NS$
14   : N->rE;N                           rl;}tfi(si,si){rl;}tfi(si
NS$
14   : SAVESTATE:          3
…
KN$
1016: SAVESTATE:          199
1016:                                    }ovl;}
}N$
1017:                                    ovl;}
N$
1018: N->ovE;N                           ovl;}
N$
```

```
1018: SAVESTATE:          200
1018:                                ovl;}
ovE;N$
1019:                                vl;}
vE;N$
1020:                                l;}
E;N$
1021: E->lvE                         l;}
E;N$
1021: SAVESTATE:          201
1021:                                l;}
lvE;N$
1022:                                ;}
vE;N$
1023: TS_NOK / NS_NORULECHAIN
1023: RESSTATE
1023:                                l;}
E;N$
1024: E->l                           l;}
E;N$
1024: SAVESTATE:          201
1024:                                l;}
l;N$
1025:                                ;}
;N$
1026:                                }
N$
1027: N->}                           }
N$
1027: SAVESTATE:          202
1027:                                }
}$
1028:                                                                $
1029: LENTA_END
1030: ------>LENTA_END
```

Листинг 4 – Разбор исходного кода синтаксическим анализатором

Приложение Г

```
namespace Polish {
     int GetPriority(LT::Entry entry) {
          if (entry.lexema == LEX_EQUAL) return 0;
          if (entry.lexema == LEX_OPERATOR || entry.lexema ==
LEX_LOGOPERATOR) {
               switch (entry.op) {
               case LT::OEQ: case LT::ONE: return 1;
               case LT::OMORE: case LT::OLESS: case LT::OGE: case
LT::OLE: return 1;
               case LT::OLESS_CMP: return 1;
               case LT::OPLUS: case LT::OMINUS: return 2;
               case LT::OMUL: case LT::ODIV: case LT::OMOD:
return 3;
               default: return 0;
               }
          }
          return -1;
     }

     bool PolishNotation(int i, Lex::LEX& lex, char terminator) {
          std::stack<LT::Entry> stack;
          std::vector<LT::Entry> out;

          // Стек для вложенных вызовов
          std::stack<std::vector<std::vector<LT::Entry>>>
funcArgsStack;
          std::stack<int> funcIdStack;

          int startPos = i;
          int itemsProcessed = 0;

          for (; i < lex.lextable.size; i++, itemsProcessed++) {
               LT::Entry t = lex.lextable.table[i];

               if (t.lexema == terminator) {
                    if (terminator == LEX_RIGHTTHESIS) {
                         if (stack.empty() || stack.top().lexema
!= LEX_LEFTTHESIS) break;
                    }
                    else {
                         break;
                    }
               }
               if (t.lexema == LEX_LEFTBRACE) break;
               bool inFunction = !funcArgsStack.empty();

               switch (t.lexema) {
               case LEX_ID:
               case LEX_LITERAL:
                    // Начало вызова функции: ID + (
```

```
                    if (t.lexema == LEX_ID && i + 1 <
lex.lextable.size && lex.lextable.table[i + 1].lexema ==
LEX_LEFTTHESIS) {
                         funcIdStack.push(t.idxTI);
                         // Новый уровень аргументов
                         std::vector<std::vector<LT::Entry>>
args;

     args.push_back(std::vector<LT::Entry>());
                         funcArgsStack.push(args);
                    }
                    else {
                         if (inFunction) {
                              if (!funcArgsStack.empty() &&
!funcArgsStack.top().empty()) {

     funcArgsStack.top().back().push_back(t);
                              }
                         }
                         else {
                              out.push_back(t);
                         }
                    }
                    break;
               case LEX_LEFTTHESIS:
                    stack.push(t);
                    break;
               case LEX_RIGHTTHESIS:
                    while (!stack.empty() && stack.top().lexema
!= LEX_LEFTTHESIS) {
                         LT::Entry top = stack.top();
stack.pop();
                         if (inFunction && !funcArgsStack.empty()
&& !funcArgsStack.top().empty())

     funcArgsStack.top().back().push_back(top);
                         else out.push_back(top);
                    }
                    if (!stack.empty()) stack.pop();
                    if (!funcIdStack.empty() && (stack.empty() ||
stack.top().lexema != LEX_LEFTTHESIS)) {
                         if (funcArgsStack.empty()) break;

                         auto args = funcArgsStack.top();
                         funcArgsStack.pop();
                         int idIdx = funcIdStack.top();
                         funcIdStack.pop();
                         bool stillInFunc =
!funcArgsStack.empty();
                         int argCount = 0;
                         for (auto& arg : args) {
                              if (!arg.empty()) argCount++;
                              for (auto& token : arg) {
```

```
                                    if (stillInFunc &&
!funcArgsStack.empty() && !funcArgsStack.top().empty())

     funcArgsStack.top().back().push_back(token);
                                    else
                                         out.push_back(token);
                               }
                          }
                          LT::Entry callOp;
                          callOp.lexema = '@';
                          callOp.sn = t.sn;
                          callOp.idxTI = idIdx;
                          callOp.priority = argCount;
                          if (stillInFunc &&
!funcArgsStack.empty() && !funcArgsStack.top().empty())

     funcArgsStack.top().back().push_back(callOp);
                          else
                               out.push_back(callOp);
                     }
                     break;
                case LEX_COMMA:
                     while (!stack.empty() && stack.top().lexema
!= LEX_LEFTTHESIS) {
                          LT::Entry top = stack.top();
stack.pop();
                          if (inFunction && !funcArgsStack.empty()
&& !funcArgsStack.top().empty())

     funcArgsStack.top().back().push_back(top);
                          else out.push_back(top);
                     }
                     if (inFunction && !funcArgsStack.empty()) {

     funcArgsStack.top().push_back(std::vector<LT::Entry>());
                     }
                     break;
                case LEX_OPERATOR:
                case LEX_LOGOPERATOR:
                case LEX_EQUAL:
                     while (!stack.empty() && stack.top().lexema
!= LEX_LEFTTHESIS &&
                          GetPriority(t) <=
GetPriority(stack.top())) {
                          LT::Entry top = stack.top();
stack.pop();
                          if (inFunction && !funcArgsStack.empty()
&& !funcArgsStack.top().empty())

     funcArgsStack.top().back().push_back(top);
else out.push_back(top);
                     }
                     stack.push(t);
```

```
                    break;
               }
          }
          while (!stack.empty()) {
               out.push_back(stack.top());
               stack.pop();
          }
          for (size_t k = 0; k < out.size(); k++) {
               lex.lextable.table[startPos + k] = out[k];
          }
          for (size_t k = out.size(); k < (size_t)itemsProcessed;
k++) {
               lex.lextable.table[startPos + k] = { '#', -1, -1
};
          }
          return true;
     }
bool StartPolish(Lex::LEX& lex) {
          for (int i = 0; i < lex.lextable.size; i++) {
               if (lex.lextable.table[i].lexema == LEX_EQUAL ||
                    lex.lextable.table[i].lexema == LEX_RETURN ||
                    lex.lextable.table[i].lexema == LEX_COUT)
               {
PolishNotation(i + 1, lex, LEX_SEMICOLON);
               }
else if (lex.lextable.table[i].lexema == LEX_SWITCH) {
                    if (i + 2 < lex.lextable.size)
                         PolishNotation(i + 2, lex,
LEX_RIGHTTHESIS);
               }
else if (lex.lextable.table[i].lexema == LEX_ID) {
                    if (i + 1 < lex.lextable.size &&
lex.lextable.table[i + 1].lexema == LEX_LEFTTHESIS) {
                         bool isProc = false;
                         if (i == 0) isProc = true;
                         else {
                              char prev = lex.lextable.table[i -
1].lexema;
                              if (prev == LEX_SEMICOLON || prev
== LEX_LEFTBRACE || prev == LEX_BRACELET || prev ==
LEX_TWOPOINT)
                                   isProc = true;
                         }
                         if (isProc) PolishNotation(i, lex,
LEX_SEMICOLON);}}}
          return true;}
```

Листинг 1 – Программная реализация механизма преобразования в ПОЛИЗ

Приложение Д

Листинг 1 – Результат генерации кода контрольного примера в Ассемблере

```
.586
.model flat, stdcall
includelib kernel32.lib
includelib msvcrtd.lib
includelib ucrtd.lib
includelib vcruntimed.lib
includelib legacy_stdio_definitions.lib
includelib "D:\Программирование\3_сем\КПО\ZMI-
2025\Debug\InLib.lib"
ExitProcess PROTO :DWORD
outnum PROTO :SDWORD
outstr PROTO :DWORD
newline PROTO
strtoint PROTO :DWORD
stcmp PROTO :DWORD, :DWORD
strle PROTO :DWORD
mabs PROTO :SDWORD
rnd PROTO :SDWORD
.stack 4096
.const
L2                                                  sdword 0
L19                                                 sdword 2
L30                                                 db 'Типы данных',
0
L32                                                 db 'Строковый тип
данных', 0
L35                                                 sdword 100
L37                                                 db 'Decimal
(100):', 0
L40                                                 sdword 255
L42                                                 db 'Hex (0xFF ->
-1):', 0
L45                                                 sdword 5
L47                                                 db 'Binary
(0b101):', 0
L50                                                 byte 'Z'
L52                                                 db 'Char (Z ->
90):', 0
L55                                                 sdword 1
L57                                                 db 'Bool (true ->
1):', 0
L60                                                 db 'Math |
Математика', 0
L63                                                 sdword 3
L65                                                 db '100 / 3 =', 0
L70                                                 db '100 % 3 =', 0
L73                                                 sdword 10
L74                                                 sdword 5
```

```
L76                                                 db 'calc(10, 5) -
> (10+5) * 2 =', 0
L80                                                 sdword -50
L82                                                 db 'mabs(-50) =',
0
L85                                                 db 'Strings |
Строки ', 0
L87                                                 db 'Привет', 0
L89                                                 db 'Length of
Privet (expect 6):', 0
L92                                                 db '42', 0
L94                                                 db 'String 42 to
Byte:', 0
L97                                                 db 'Comparison |
Сравнение ', 0
L101                                                db '10 == 10
(Expect 1):', 0
L106                                                db '10 != 5
(Expect 1):', 0
L109                                                sdword 20
L112                                                db '20 > 10
(Expect 1):', 0
L117                                                db '5 < 10
(Expect 1):', 0
L122                                                db '10 <= 10
(Expect 1):', 0
L127                                                db '5 >= 10
(Expect 0):', 0
L130                                                db 'Conditional
operator | Условный оператор', 0
L133                                                db 'Random (0 -
2):', 0
L136                                                db 'Case 0:
Zero/Ноль', 0
L138                                                db 'Case 1:
One/Один', 0
L140                                                db 'Default/По
умолчанию', 0
L142                                                db 'The end |
Конец', 0
.data
switch_val dd 0
calcres                                             sbyte 0
welcome                                             dd 0
numDec                                              sbyte 0
numHex                                              sbyte 0
numBin                                              sbyte 0
myChar                                              sbyte 0
myBool                                              sbyte 0
res                                                 sbyte 0
lenVal                                              sbyte 0
rndVal                                              sbyte 0
.code
```

```
calc PROC, a :DWORD, b :DWORD
; Decl: calcres
push a
push b
pop ebx
pop eax
add eax, ebx
push eax
push 2
pop ebx
pop eax
imul eax, ebx
push eax
pop eax
mov calcres, al
movsx eax, calcres
push eax
pop eax
ret
calc ENDP
main PROC
; Decl: welcome
; Decl: numDec
; Decl: numHex
; Decl: numBin
; Decl: myChar
; Decl: myBool
; Decl: res
; Decl: lenVal
; Decl: rndVal
push offset L30
pop eax
mov welcome, eax
push offset L32
pop eax
mov welcome, eax
push welcome
pop eax
invoke outstr, eax
invoke newline
push 100
pop eax
mov numDec, al
push offset L37
pop eax
invoke outstr, eax
invoke newline
movsx eax, numDec
push eax
pop eax
invoke outnum, eax
invoke newline
push 255
```

```
pop eax
mov numHex, al
push offset L42
pop eax
invoke outstr, eax
invoke newline
movsx eax, numHex
push eax
pop eax
invoke outnum, eax
invoke newline
push 5
pop eax
mov numBin, al
push offset L47
pop eax
invoke outstr, eax
invoke newline
movsx eax, numBin
push eax
pop eax
invoke outnum, eax
invoke newline
push 90
pop eax
mov myChar, al
push offset L52
pop eax
invoke outstr, eax
invoke newline
movsx eax, myChar
push eax
pop eax
invoke outnum, eax
invoke newline
push 1
pop eax
mov myBool, al
push offset L57
pop eax
invoke outstr, eax
invoke newline
movsx eax, myBool
push eax
pop eax
invoke outnum, eax
invoke newline
push offset L60
pop eax
invoke outstr, eax
invoke newline
push 100
push 3
```

```
pop ebx
pop eax
cdq
idiv ebx
push eax
pop eax
mov res, al
push offset L65
pop eax
invoke outstr, eax
invoke newline
movsx eax, res
push eax
pop eax
invoke outnum, eax
invoke newline
push 100
push 3
pop ebx
pop eax
cdq
idiv ebx
mov eax, edx
push eax
pop eax
mov res, al
push offset L70
pop eax
invoke outstr, eax
invoke newline
movsx eax, res
push eax
pop eax
invoke outnum, eax
invoke newline
push 10
push 5
call calc
push eax
pop eax
mov res, al
push offset L76
pop eax
invoke outstr, eax
invoke newline
movsx eax, res
push eax
pop eax
invoke outnum, eax
invoke newline
push -50
call mabs
push eax
```

```
pop eax
mov res, al
push offset L82
pop eax
invoke outstr, eax
invoke newline
movsx eax, res
push eax
pop eax
invoke outnum, eax
invoke newline
push offset L85
pop eax
invoke outstr, eax
invoke newline
push offset L87
call strle
push eax
pop eax
mov lenVal, al
push offset L89
pop eax
invoke outstr, eax
invoke newline
movsx eax, lenVal
push eax
pop eax
invoke outnum, eax
invoke newline
push offset L92
call strtoint
push eax
pop eax
mov res, al
push offset L94
pop eax
invoke outstr, eax
invoke newline
movsx eax, res
push eax
pop eax
invoke outnum, eax
invoke newline
push offset L97
pop eax
invoke outstr, eax
invoke newline
push 10
push 10
pop ebx
pop eax
cmp eax, ebx
mov eax, 0
```

```
sete al
push eax
pop eax
mov res, al
push offset L101
pop eax
invoke outstr, eax
invoke newline
movsx eax, res
push eax
pop eax
invoke outnum, eax
invoke newline
push 10
push 5
pop ebx
pop eax
cmp eax, ebx
mov eax, 0
setne al
push eax
pop eax
mov res, al
push offset L106
pop eax
invoke outstr, eax
invoke newline
movsx eax, res
push eax
pop eax
invoke outnum, eax
invoke newline
push 20
push 10
pop ebx
pop eax
cmp eax, ebx
mov eax, 0
setg al
push eax
pop eax
mov res, al
push offset L112
pop eax
invoke outstr, eax
invoke newline
movsx eax, res
push eax
pop eax
invoke outnum, eax
invoke newline
push 5
push 10
```

```
pop ebx
pop eax
cmp eax, ebx
mov eax, 0
setl al
push eax
pop eax
mov res, al
push offset L117
pop eax
invoke outstr, eax
invoke newline
movsx eax, res
push eax
pop eax
invoke outnum, eax
invoke newline
push 10
push 10
pop ebx
pop eax
cmp eax, ebx
mov eax, 0
setle al
push eax
pop eax
mov res, al
push offset L122
pop eax
invoke outstr, eax
invoke newline
movsx eax, res
push eax
pop eax
invoke outnum, eax
invoke newline
push 5
push 10
pop ebx
pop eax
cmp eax, ebx
mov eax, 0
setge al
push eax
pop eax
mov res, al
push offset L127
pop eax
invoke outstr, eax
invoke newline
movsx eax, res
push eax
pop eax
```

```
invoke outnum, eax
invoke newline
push offset L130
pop eax
invoke outstr, eax
invoke newline
push 3
call rnd
push eax
pop eax
mov rndVal, al
push offset L133
pop eax
invoke outstr, eax
invoke newline
movsx eax, rndVal
push eax
pop eax
invoke outnum, eax
invoke newline
pop eax
mov switch_val, eax
movsx eax, rndVal
push eax
mov eax, switch_val
cmp eax, 0
jne sw_397_next_0
push offset L136
pop eax
invoke outstr, eax
invoke newline
jmp sw_397_end
sw_397_next_0:
mov eax, switch_val
cmp eax, 1
jne sw_397_next_1
push offset L138
pop eax
invoke outstr, eax
invoke newline
jmp sw_397_end
sw_397_next_1:
push offset L140
pop eax
invoke outstr, eax
invoke newline
sw_397_end:
push offset L142
pop eax
invoke outstr, eax
invoke newline
invoke ExitProcess, 0
main ENDP
```