МИНИСТЕРСТВО ОБРАЗОВАНИЯ РЕСПУБЛИКИ БЕЛАРУСЬ

Учреждение образования «БЕЛОРУССКИЙ ГОСУДАРСТВЕННЫЙ ТЕХНОЛОГИЧЕСКИЙ УНИВЕРСИТЕТ»

Факультет информационных технологий
Кафедра программной инженерии
Специальность 6-05-0612-01 Программная инженерия

ПОЯСНИТЕЛЬНАЯ ЗАПИСКА К КУРСОВОМУ ПРОЕКТУ НА ТЕМУ:

«Разработка компилятора BDV-2025»

Выполнил студент _____ Бакуменко Данила Владимирович
курс, группа подпись (Ф.И.О. студента)

Руководитель проекта _____ ассистент Волчек Дарья Ивановна
(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 _____ ассистент Волчек Дарья Ивановна
(учен. степень, звание, должность, подпись, Ф.И.О.)

Курсовой проект защищен с оценкой _____

Минск 2025

Содержание

Введение

Целью данного курсового проекта является разработка компилятора для языка программирования BDV-2025, предназначенного для выполнения простейших операций над беззнаковыми целыми числами и строками. Компилятор написан на языке C++ в среде Visual Studio 2022 под Windows 11 (24H2, 64-bit).

Исходный код на языке BDV-2025 транслируется в язык ассемблера.

Компилятор состоит из следующих частей:

– Лексический анализатор.

– Синтаксический анализатор.

– Семантический анализатор.

– Генератор кода.

Для разработки компилятора необходимо выполнить следующие задачи:

– Разработать спецификацию языка.

– Разработать структуру транслятора.

– Разработать лексический, синтаксический и семантический анализаторы.

– Разработать алгоритм преобразования выражений.

– Разработать алгоритм генерации кода на язык ассемблера.

– Провести 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 BDV-2025 является процедурным, универсальным, строго типизированным, не объектно-ориентированным, компилируемым.

1.2 Определение алфавита языка программирования

В алфавит языка BDV-2025 входят символы латиницы [a–z, A–Z], цифры [0–9], знаки операций (+ - * / % ^ & > < ! =) и разделители(( ) { } , ; : “ пробел). Полная таблица входных символов приведена в п. 3.2.

1.3 Применяемые сепараторы

Символы-сепараторы – это программные конструкции, служащие для синтаксического разделения лексем и блоков кода. Они представлены в таблице 1.1.

Таблица 1.1 – Символы-сепараторы языка BDV-2025

Сепаратор	Название	Область применения
{…}	Фигурные скобки	Обозначение границ программного блока
(…)	Круглые скобки	Выделение списка параметров функции, управление приоритетом операций, выделение выражений в управляющих конструкциях (switch, while)
,	Запятая	Разделение параметров в объявлениях и вызовах функций
;	Точка с запятой	Обозначение конца оператора
:	Двоеточие	Отделение метки (case/default) от инструкций
" … "	Двойные кавычки	Ограничение строковых литералов

1.4 Применяемые кодировки

Для написания программ на языке BDV-2025 используется базовая 7-битная кодировка ASCII. Транслятор обрабатывает входные данные как поток байтов, но интерпретирует только символы со кодами 0–127, соответствующие стандарту ASCII. Все символы с кодами 128–255 считаются недопустимыми, что гарантирует простоту и переносимость лексического анализа, исключая необходимость поддержки каких-либо 8-битных расширений ASCII (таких как Windows-1251 или KOI8-R) или сложного декодирования многобайтовых последовательностей Unicode. Это позволяет считывать лексемы непосредственно в объекты строкового типа, упрощая внутреннюю логику работы со строковыми литералами и идентификаторами. Таблица допустимых символов (ASCII 0-127) представлена на рисунке 1.1.

	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
00	00 □	01 □	02 □	03 □	04 □	05 □	06 □	07 □	08 □	09 □	0A □	0B □	0C □	0D □	0E □	0F □
10	10 □	11 □	12 □	13 □	14 □	15 □	16 □	17 □	18 □	19 □	1A □	1B □	1C □	1D □	1E □	1F □
20	20	21 !	22 “	23 #	24 $	25 %	26 &	27 (	28)	29 '	2A *	2B +	2C ,	2D -	2E .	2F /
30	30 0	31 1	32 2	33 3	34 4	35 5	36 6	37 7	38 8	39 9	3A :	3B ;	3C <	3D =	3E >	3F ?
40	40 @	41 A	42 B	43 C	44 D	45 E	46 F	47 G	48 H	49 I	4A J	4B K	4C L	4D M	4E N	4F O
50	50 P	51 Q	52 R	53 S	54 T	55 U	56 V	57 W	58 X	59 Y	5A Z	5B [	5C \	5D]	5E ^	5F _
60	60 `	61 a	62 b	63 c	64 d	65 e	66 f	67 g	68 h	69 i	6A j	6B k	6C l	6D m	6E o	6F p
70	70 p	71 q	72 r	73 s	74 t	75 u	76 v	77 w	78 x	79 y	7A z	7B {	7C |	7D }	7E ~	7F □

Рисунок 1.1 – Таблица символов кодировки ASCII

1.5 Типы данных

Тип данных – это классификация данных, определяющая допустимые значения и операции над ними. В BDV-2025 реализованы два типа: беззнаковый целый и строковый. Описание — в таблице 1.2.

Таблица 1.2 – Типы данных языка BDV-2025

Тип данных	Описание типа данных
Беззнаковый целый uint	Беззнаковый целый тип данных для хранения неотрицательных целых чисел. Занимает 4 байта памяти. Диапазон значений: от 0 до 4,294,967,295. Не может хранить отрицательные числа.
Строковый, тип данных string	Представляет собой тип данных, предназначенный для хранения последовательности символов. Может содержать буквы, цифры, пробелы и другие символы. Инициализация по умолчанию – "" (пустая строка).

1.6 Преобразование типов данных

Так как язык строго типизированный, то преобразование типов данных не поддерживается.

1.7 Идентификаторы

Идентификаторы начинаются с латинской буквы и могут содержать буквы и цифры; максимальная длина — 32 символа. Совпадения с ключевыми словами и именами стандартных функций запрещены. Префиксы по области видимости не используются. Количество уникальных идентификаторов ограничено таблицей (4096 записей).

1.8 Литералы

С помощью литералов инициализируются переменные. В BDV-2025 есть три вида: целочисленные (десятичные и шестнадцатеричные) и строковые. Символьных литералов нет. Описание — в таблице 1.3.

Таблица 1.3 – Описание литералов языка BDV-2025

Тип литерала	Описание литерала
Беззнаковый целочисленный литерал в десятичном представлении	Беззнаковое десятичное число. Диапазон: 0 … 4 294 967 295; Литерал может быть только rvalue.
Беззнаковый целочисленный литерал в шестнадцатеричном представлении	Беззнаковое шестнадцатеричное число с обязательным префиксом 0x, далее цифры 0–9 и буквы a–f/A–F. Значение должно помещаться в uint. Данный литерал также является только rvalue.
Строковый литерал	Строковый литерал состоит из последовательности символов латинского алфавита и цифр, заключенных в двойные кавычки. Данный литерал является rvalue.

1.9 Объявление данных

Для объявления переменной используется ключевое слово adv, после которого указывается тип данных (uint или string) и имя идентификатора. Допускается инициализация при объявлении. Для объявления функций используется ключевое слово function, перед которым указывается тип возвращаемого значения (uint или string). После — имя функции. Далее обязателен список параметров (каждый с типом) и тело функции. Рекурсия не допускается.

1.10 Инициализация данных

Объектами-инициализаторами могут быть только идентификаторы или литералы того же типа. Значения по умолчанию: 0 для uint, пустая строка "" для string. Инициализация при объявлении допускается.

1.11 Инструкции языка

Инструкции языка BDV-2025 отображены в таблице 1.4.

Таблица 1.4 – Инструкции языка программирования BDV-2025

Инструкция	Запись на языке BDV-2025
Объявление переменной	adv <тип данных> <идентификатор>;
Объявление функции	<тип данных> function <идентификатор> (<тип данных> <идентификатор>, …) {… return <выражение>;};
Присваивание	<идентификатор> = <выражение>;
Возврат из подпрограммы	return <выражение>;
Вывод данных	cout <выражение>;
Вызов функции или процедуры	<идентификатор функции> (<список параметров>);
Условный выбор	switch(<выражение>) { case <литерал>: … default: … }
Цикл	while(<выражение>) { … }

1.12 Операции языка

В языке BDV-2025 предусмотрено несколько операций над данными. Все операции, с которыми можно работать представлены в таблице 1.5.

Таблица 1.5 – Операции языка программирования BDV-2025

Тип оператора	Оператор
Арифметические	1. + – сложение 2. - – вычитание 3. * – умножение 4. / – деление нацело 5. % – определение остатка от деления 6. = – присваивание 7. ^ – возведение в степень 8. & – возведение в квадрат
Строковые	1. = – присваивание
Сравнение	1. > – больше 2. < – меньше 3. != – не равно 4. <= – меньше или равно 5. >= – больше или равно

1.13 Выражения и их вычисление

Выражения записываются в одну строку, приоритет операций можно менять скобками. Два оператора подряд не допускаются. В выражениях разрешён вызов функций, возвращающих значение требуемого типа (uint или string). Смешение типов недопустимо. Арифметические операции и операции сравнения определены только для типа uint. Операции над строками (в том числе конкатенация) осуществляются исключительно через вызов функций стандартной библиотеки. Перед генерацией кода каждое выражение приводится к польской нотации для упрощения вычисления в ассемблере.

1.14 Конструкции языка

Программа BDV-2025 состоит из пользовательских функций и главной функции main. Конструкции оформляются в фигурных скобках; рекомендуется использовать отступы для читаемости. Программные конструкции приведены в таблице 1.6.

Таблица 1.6 – Программные конструкции языка BDV-2025

Конструкция	Реализация

Главная функция	main
	{
	<программный блок>
	}

Функция	<тип данных> function <идентификатор>(<тип данных> <идентификатор>, …)
	{
	…
	return <выражение >;
	}

Цикл	while(<uint-выражение>) { … }
Условный выбор	switch(<uint-выражение>) { case <литерал>: … default: … } (после каждого case/default — неявный выход)

1.15 Область видимости идентификаторов

Область видимости блочная: идентификатор виден от места объявления до конца текущего блок {…}. Переменные и параметры функции доступны только внутри этой функции; глобальных переменных нет. Повторное объявление в одном блоке запрещено. Префиксы по имени функции не добавляются; имя хранится как есть (до 32 символов).

1.16 Семантические проверки

В языке программирования BDV-2025 есть следующие правила семантической проверки исходного текста языка, представленные в таблице 1.7.

Таблица 1.7 – Семантические проверки языка BDV-2025

Номер	Правило	Код ошибки
1	Необъявленный идентификатор	300
2	Отсутствует точка входа main	301
3	Обнаружено несколько точек входа main	302
4	Попытка переопределения идентификатора	305
5	Превышено максимальное количество параметров функции	306
6	Слишком много параметров в вызове	307
7	Кол-во ожидаемыех функцией и передаваемых параметров не совпадают	308

Продолжение таблицы 1.7

8	Несовпадение типов передаваемых параметров	309
9	Обнаружен символ \'\"\'. Возможно, не закрыт строковый литерал	311
10	Превышен размер строкового литерала	312
11	Недопустимый целочисленный литерал	313
12	Типы данных в выражении не совпадают	314
13	Семантическая ошибка: Недопустимое строковое выражение справа от знака \'=\'	316
14	Деление на ноль	318
15	Оператор ^ требует операнды типа uint	319
16	Выражение в switch должно быть типа uint	320
17	Значение case должно быть целочисленным литералом	321
18	Дублирование значения case	322
19	switch должен содержать хотя бы один case	323
20	Hex-литерал выходит за пределы uint	325
21	Операции сравнения требуют операнды одного типа	326
22	Рекурсия не поддерживается	328
23	Функция должна вызываться со скобками ()	329

Назначение семантического анализа – проверка смысловой правильности конструкций языка программирования.

1.17 Распределение оперативной памяти на этапе выполнения

Транслированный код использует сегмент констант (.const) для всех литералов и сегмент данных (.data) для переменных и параметров. Локальность обеспечивается на уровне исходного языка блочной областью видимости. В сгенерированном ассемблере каждому идентификатору сопоставляется уникальная метка памяти (генерируемая на основе имени и индекса в таблице символов), что позволяет эмулировать локальную область видимости при использовании статического распределения памяти.

1.18 Стандартная библиотека и её состав

В языке BDV-2025 присутствует набор встроенных функций, которые могут быть вызваны из исходного кода. Эти функции обеспечивают базовые операции над строками и преобразования типов. Состав встроенных функций представлен в таблице 1.8.

Таблица 1.8 – Основные функции стандартной библиотеки языка BDV-2025

Функция	Описание
stringToUnsigned(string s)	Преобразует строку s в беззнаковое целое (uint). Поддерживает десятичный и шестнадцатеричный (префикс 0x) форматы, игнорирует начальные пробелы.

Продолжение таблицы 1.8

unsignedToString(uint x)	Преобразует беззнаковое целое x в строку (десятичное представление).
stringConcat(string a, string b)	Возвращает новую строку, являющуюся конкатенацией строк a и b (результирующая длина ограничена 255 символами).
stringLength(string s)	Возвращает длину строки s (тип uint).

Эти функции доступны программисту на языке BDV-2025 и вызываются как обычные функции. Их реализация обеспечивается статической библиотекой на C++, которая подключается на этапе генерации кода. Служебный параметр (буфер для результата), присутствующий в реализации библиотеки, передается транслятором автоматически и скрыт от пользователя.

1.19 Ввод и вывод данных

Вывод выполняется оператором cout; можно выводить литералы и идентификаторы типов uint и string. В сгенерированном коде cout вызывает внутренние C++-функции outUint (для чисел) и outString (для строк) из статической библиотеки, подключаемой на этапе генерации. Прямого ввода в языке нет.

1.20 Точка входа

В языке BDV-2025 каждая программа должна содержать главную функцию (точку входа) main. Данная функция может быть определена в программе только один раз и не может встречаться более одного раза или отсутствовать вообще. В случае нарушения данных условий будет зарегистрирована ошибка.

1.21 Препроцессор

В языке программирования BDV-2025 препроцессор отсутствует.

1.22 Соглашение о вызовах

В языке программирования BDV-2025 используется соглашение о вызовах stdcall. Аргументы функций передаются через стек (инструкция push) в порядке справа налево. Очистку стека от аргументов производит вызываемая функция (ret N). Результат работы функции возвращается через регистр EAX.

1.23 Объектный код

Код на языке программирования BDV-2025 транслируется в исходный код на языке ассемблера.

1.24 Классификация сообщений транслятора

В случае возникновении ошибки в коде программы, в файл протокола будет выведено сообщение об ошибке с указанием места встречи этой ошибки. Классификация ошибок представлена в таблице 1.11.

Таблица 1.9 – Классификация ошибок

Номера ошибок	Характеристика
0 – 99	Системные ошибки
100 – 103	Ошибки входных параметров
200 – 205 и 327	Лексические ошибки
300 – 326, 328, 329	Семантические ошибки
600 – 609	Синтаксические ошибки

Обрабатываются ошибки на всех этапах обработки исходного кода, то есть во время прохождения различных этапов анализа.

1.25 Контрольный пример

Контрольный пример показывает работу всех функций и показывает особенности языка BDV-2025. Исходный код контрольного примера представлен в приложении А.

2 Структура транслятора

2.1 Компоненты транслятора, их назначение и принципы взаимодействия

В BDV-2025 исходный код транслируется в ассемблер (MASM синтаксис), далее собирается компилятором MASM. Транслятор состоит из стадий (лексический, синтаксический, семантический анализаторы, преобразователь в польскую запись и генератор кода), которые последовательно обмениваются таблицей лексем и таблицей идентификаторов. Параметры запуска задают пути выходных файлов ТЛ/ТИ/ASM. Структура транслятора показана на рис. 2.1.

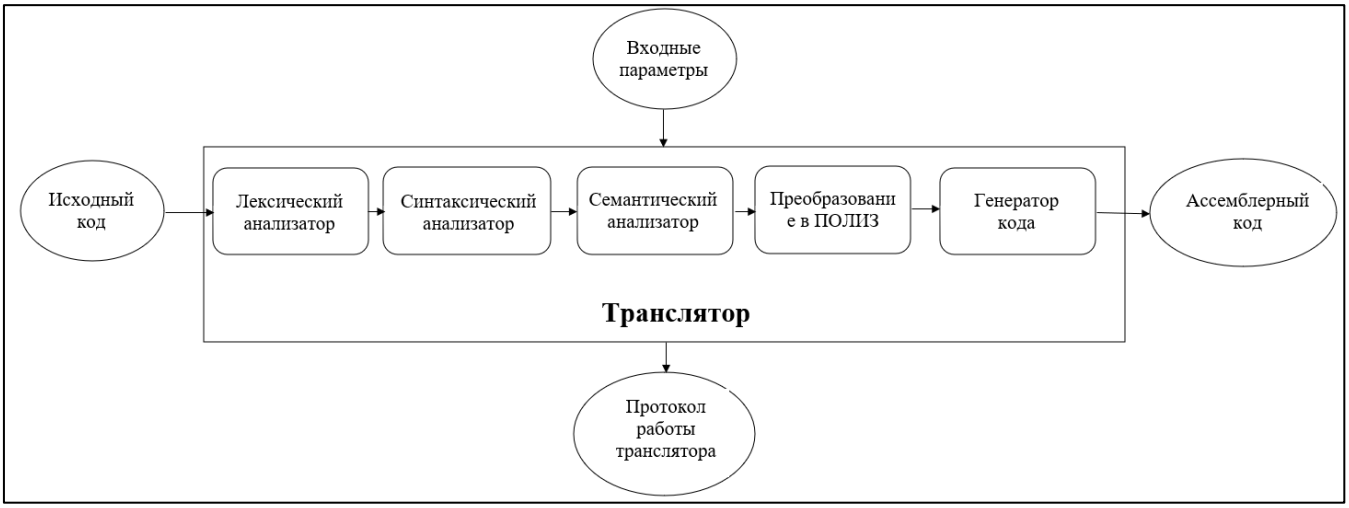


Рисунок 2.1 – Структура транслятора языка BDV-2025

Первой стадией работы компилятора является Лексический анализ, выполняемый лексическим анализатором (сканером). На вход этого модуля подаётся последовательность символов исходного кода. Сканер выполняет чтение файла, фильтрацию «мусора» (удаление пробелов, табуляций, комментариев) и первичную группировку символов в значащие единицы — лексемы. Реализация данного этапа базируется на конечных автоматах (FST), где для каждого класса лексем построен свой граф переходов. Результатом работы является формирование Таблицы Лексем (ТЛ) и наполнение Таблицы Идентификаторов (ТИ), которые служат входными данными для следующих этапов.

Следующим этапом выступает Синтаксический анализ, выполняемый парсером. Он отвечает за проверку грамматической правильности программы и соблюдение порядка следования лексем. В данном проекте анализатор реализован на основе автомата с магазинной памятью (МП-автомат). Для анализа используется контекстно-свободная грамматика языка, приведенная к нормальной форме Грейбах (НФГ), что позволяет выполнять разбор сверху-вниз, используя стек для контроля вложенности конструкций и баланса скобок.

После успешного разбора запускается Семантический анализатор, выполняющий проверку смысловой корректности программы. Анализатор проходит по сформированным таблицам, контролируя типы данных и области видимости переменных. На этом этапе выявляются ошибки, которые невозможно обнаружить

синтаксически, например, попытки выполнения операций над несовместимыми типами данных или использование необъявленных идентификаторов.

Важным промежуточным этапом является Преобразование в Польскую нотацию (ПОЛИЗ). Этот модуль преобразует выражения из стандартной инфиксной записи в постфиксную форму (Обратная Польская Запись). Данное преобразование необходимо для линеаризации арифметических и логических выражений, а также для преобразования управляющих конструкций (циклов, условий) в последовательность меток и переходов, что значительно упрощает дальнейшую генерацию машинных инструкций.

Завершает процесс трансляции Генератор кода. Он принимает на вход таблицы с данными в формате ПОЛИЗ и преобразует их в результирующий код на языке ассемблера. Генератор размещает переменные в сегменте данных, формирует сегмент кода и реализует логику вычислений через стек процессора. Итогом работы всего транслятора является файл с расширением .asm, готовый для сборки компоновщиком.

2.2 Перечень параметров транслятора

Для управления процессом трансляции и настройки путей выходных файлов используются параметры командной строки. Гибкая конфигурация позволяет интегрировать транслятор в автоматизированные сценарии сборки (batch-скрипты) или среды разработки. Перечень доступных ключей запуска приведен в таблице 2.1.

Таблица 2.1 – Входные параметры транслятора языка BDV-2025

Входной параметр	Описание параметра	Значение по умолчанию
-in:<путь к in-файлу>	Полный или относительный путь к файлу с исходным кодом.	Обязательный параметр
-out:<путь>	Путь к выходному файлу ассемблера (.asm).	<имя_in>.asm
-log:<путь к log-файлу>	Путь к файлу журнала (.log) для вывода протоколов анализа.	<имя_in>.log

2.3 Протоколы, формируемые транслятором

В процессе трансляции исходного кода программный комплекс BDV-2025 генерирует структурированный набор выходных файлов, отражающих результаты работы различных этапов компиляции. Формирование этих протоколов является критически важным для обеспечения наблюдаемости процесса трансляции, позволяя разработчику контролировать корректность разбора кода и диагностировать возникающие ошибки.

По функциональному назначению выходные данные можно разделить на две категории. К первой относятся диагностические протоколы, центральным элементом которых является файл журнала. Он создается на этапе инициализации и остается открытым для записи до завершения работы всех модулей. В журнал

последовательно заносятся: параметры запуска, содержимое Таблицы Лексем (ТЛ) и Таблицы Идентификаторов (ТИ), результаты работы парсера и семантического анализатора. Особое место занимает вывод промежуточного представления кода (ПОЛИЗ), который позволяет отследить корректность преобразования выражений.

Ко второй категории относится целевой код. Результатом успешной трансляции является генерация программного кода в виде текстового файла на языке ассемблера. Файл содержит сегмент данных (.data) с инициализированными переменными и буферами, а также сегмент кода (.code) с инструкциями процессора и вызовами процедур. Сгенерированный файл полностью готов к нативной компиляции и линковке с системными библиотеками.

Полный перечень и техническое описание формируемых файлов приведены в таблице 2.2.

Таблица 2.2 – Протоколы, формируемые транслятором языка BDV-2025

Формируемый протокол	Описание выходного протокола
Файл журнала (.log)	Файл с протоколом работы транслятора. Содержит подробную информацию: таблицу лексем (ТЛ), таблицу идентификаторов (ТИ), протокол синтаксического анализа, дерево разбора, листинг польской нотации (ПОЛИЗ) и сообщения об ошибках.
Выходной файл (.asm)	Результат работы программы – файл, содержащий исходный код на языке ассемблера (MASM), готовый к компиляции.

Таким образом, реализованная система выходных данных обеспечивает полную прозрачность процесса трансляции. Детальное протоколирование в файле журнала позволяет разработчику отслеживать корректность разбора на каждом этапе и быстро локализовать ошибки. Сгенерированный ассемблерный файл, в свою очередь, является стандартным исходным кодом для MASM, что гарантирует совместимость с системными инструментами разработки и возможность дальнейшего получения исполняемого модуля.

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. Основная задача модуля — выделить в потоке символов простейшие конструкции языка и сформировать внутренние структуры данных. Алгоритм работы анализатора в проекте BDV-2025 построен на базе конечных автоматов (FST), где входная последовательность символов обрабатывается в соответствии с графом переходов, что позволяет однозначно классифицировать каждую лексему.

Одним из ключевых действий лексического анализатора является предварительная очистка входного потока. Модуль выполняет удаление «пустых» символов (пробелов, знаков табуляции, переводов строк) и комментариев. Если эти элементы будут удалены на данном этапе, синтаксический анализатор никогда не столкнется с ними, что существенно упрощает разработку грамматики и логику дальнейшего разбора.

В процессе разбора анализатор производит токенизацию — группировку символов в значащие лексические единицы, разделяя их на идентификаторы, ключевые слова, строковые и числовые литералы, а также разделители и знаки операций. Результатом этой классификации становится заполнение Таблицы Лексем (ТЛ) и первичное наполнение Таблицы Идентификаторов (ТИ). Каждой лексеме сопоставляется её тип, внутреннее представление, а также номер строки и позиция в коде для последующей диагностики ошибок.

Исходный код программы представлен в приложении А, структура лексического анализатора — на рисунке 3.1. В результате работы данного этапа исходный текст полностью преобразуется в промежуточное представление (последовательность лексем), которое служит входными данными для следующей фазы компилятора.

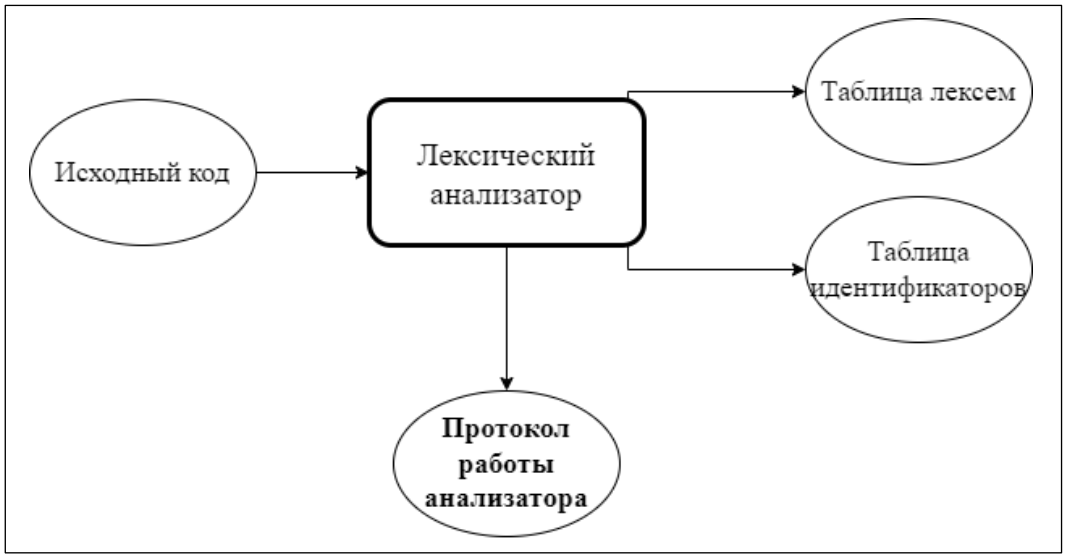


Рисунок 3.1 – Структура лексического анализатора

Такое разделение выходных данных на два независимых потока (таблицу лексем и таблицу идентификаторов) позволяет оптимизировать дальнейшую обработку. Таблица лексем, представляющая собой линеаризованный код программы, служит непосредственным входом для синтаксического анализатора, обеспечивая высокую скорость проверки грамматики. Таблица идентификаторов, в свою очередь, исключает дублирование строковых данных и используется преимущественно на этапе семантического анализа и генерации кода для контроля типов и выделения памяти.

3.2 Контроль входных символов

Для классификации символов входного потока и фильтрации недопустимых знаков используется таблица кодов ASCII, отображенная в массив категорий. Каждому символу (0-255) сопоставляется код категории, определяющий его роль в лексическом анализе. Таблица кодировки символов представлена на рисунке 3.2, а расшифровка категорий — в таблице 3.1.

```
#define IN_CODE_TABLE   {\
        IN::F, IN::F, IN::F, IN::F,  IN::F, IN::F, IN::F, IN::F,   IN::F, IN::P, IN::N, IN::F,  IN::F, IN::F, IN::F, IN::F,\
        IN::F, IN::F, IN::F, IN::F,  IN::F, IN::F, IN::F, IN::F,   IN::F, IN::F, IN::F, IN::F,  IN::F, IN::F, IN::F, IN::F,\
        IN::P, IN::S, IN::Q, IN::S,  IN::T, IN::T, IN::S, IN::Q,   IN::S, IN::S, IN::S, IN::S,  IN::S, IN::S, IN::T, IN::S,\
        IN::T, IN::T, IN::T, IN::T,  IN::T, IN::T, IN::T, IN::T,   IN::T, IN::T, IN::T, IN::S,  IN::S, IN::S, IN::S, IN::T,\
        IN::T, IN::T, IN::T, IN::T,  IN::T, IN::T, IN::T, IN::T,   IN::T, IN::T, IN::T, IN::T,  IN::T, IN::T, IN::T, IN::T,\
        IN::T, IN::T, IN::T, IN::T,  IN::T, IN::T, IN::T, IN::T,   IN::T, IN::T, IN::T, IN::S,  IN::T, IN::S, IN::T, IN::T,\
        IN::F, IN::T, IN::T, IN::T,  IN::T, IN::T, IN::T, IN::T,   IN::T, IN::T, IN::T, IN::T,  IN::T, IN::T, IN::T, IN::T,\
        IN::T, IN::T, IN::T, IN::T,  IN::T, IN::T, IN::T, IN::T,   IN::T, IN::T, IN::T, IN::S,  IN::T, IN::T, IN::S, IN::T,\
                                                                                                                        \
        IN::F, IN::F, IN::F, IN::F,  IN::F, IN::F, IN::F, IN::F,   IN::F, IN::F, IN::F, IN::F,  IN::F, IN::F, IN::F, IN::F,\
        IN::F, IN::F, IN::F, IN::F,  IN::F, IN::F, IN::F, IN::F,   IN::F, IN::F, IN::F, IN::F,  IN::F, IN::F, IN::F, IN::F,\
        IN::F, IN::F, IN::F, IN::F,  IN::F, IN::F, IN::F, IN::F,   IN::F, IN::F, IN::F, IN::F,  IN::F, IN::F, IN::F, IN::F,\
        IN::F, IN::F, IN::F, IN::F,  IN::F, IN::F, IN::F, IN::F,   IN::F, IN::F, IN::F, IN::F,  IN::F, IN::F, IN::F, IN::F,\
        IN::F, IN::F, IN::F, IN::F,  IN::F, IN::F, IN::F, IN::F,   IN::F, IN::F, IN::F, IN::F,  IN::F, IN::F, IN::F, IN::F,\
        IN::F, IN::F, IN::F, IN::F,  IN::F, IN::F, IN::F, IN::F,   IN::F, IN::F, IN::F, IN::F,  IN::F, IN::F, IN::F, IN::F,\
        IN::F, IN::F, IN::F, IN::F,  IN::F, IN::F, IN::F, IN::F,   IN::F, IN::F, IN::F, IN::F,  IN::F, IN::F, IN::F, IN::F,\
        IN::F, IN::F, IN::F, IN::F,  IN::F, IN::F, IN::F, IN::F,   IN::F, IN::F, IN::F, IN::F,  IN::F, IN::F, IN::F, IN::F \
}
```

Рисунок 3.2 – Таблица контроля входных символов

Таблица 3.1 – Соответствие символов и их значений в таблице

Значение в таблице входных символов	Символы
Разрешенный	T
Запрещенный	F
Игнорируемый	I
Литерал	Q
Сепаратор	S
Перевод строки	N
Пробел, табуляция	P

3.3 Удаление избыточных символов

Избыточными символами считаются пробелы и знаки табуляции, не находящиеся внутри строковых литералов. Эти символы служат лишь для форматирования кода и удаляются на раннем этапе разбора.

Алгоритм очистки: посимвольное чтение файла исходного кода; если встречен символ категории P (Space), он игнорируется (не добавляется в буфер лексемы), но служит сигналом завершения предыдущей лексемы (как сепаратор); символы перевода строки N учитываются для инкремента счетчика строк, но также не попадают в результирующие лексемы.

3.4 Перечень ключевых слов

Лексический анализатор преобразует исходный текст в последовательность лексем. Каждой конструкции языка (ключевому слову, оператору, идентификатору) присваивается уникальный односимвольный код (тип лексемы). Соответствие токенов и их кодов приведено в таблице 3.2.

Таблица 3.2 – Соответствие токенов и сепараторов с лексемами

Токен	Лексема	Пояснение
uint, string	t	Название типов данных.
Идентификатор	i	Длина идентификатора – 32 символов.
Литерал	l	Допустимые литералы.
Шестнадцатеричный литерал	H	Целочисленный литерал в шестнадцатеричном представлении.
adv	A	Объявление переменной
function	f	Объявление функции.
return	e	Возврат из функции.
main	m	Главная функция.
cout	o	Вывод.
switch	U	Конструкция выбора
case	C	Ветка выбора
default	D	Ветка по умолчанию
while	W	Цикл
;	;	Разделение выражений.
:	:	После case/default в switch.
,	,	Разделение параметров функции.
{	{	Начало блока/тела функции.
}	}	Закрытие блока/тела функции.
(	(	Передача параметров в функцию, приоритет операций.
)	)	Закрытие блока для передачи параметров, приоритет операций.
=	=	Знак присваивания.
+	+	Сложение.
-	-	Вычитание.
*	*	Умножение.
/	/	Деление нацело.
%	%	Остаток от деления.

Продолжение таблицы 3.2

^	^	Возведение в степень
>	>	Проверка на больше.
<	<	Проверка на меньше.
<=	@	меньше или равно
>=	~	больше или равно
==	#	равно
!=	!	Не равно
&	&	Возведение в квадрат

Каждому токену соответствует свой детерминированный конечный автомат (FST). Распознавание происходит путем подачи входной цепочки символов на граф переходов автомата. Если автомат, перемещаясь по узлам и ребрам в соответствии с входными символами, достигает конечного состояния, лексема считается распознанной и записывается в таблицу лексем. Если выражение является идентификатором или литералом, информация о нем дополнительно заносится в таблицу идентификаторов. Структура конечного автомата изображена на рисунке 3.3.

```
struct RELATION     // Ребро: символ -> следующая вершина
{
    char  symbol;   // Символ перехода (например 'a')
    short nnode;    // Номер следующей вершины
    RELATION(
        char c = 0x00,
        short ns = 0
    );
};

struct NODE         // Вершина графа
{
    short n_relation;       // Количество исходящих ребер
    RELATION* relations;    // Массив ребер
    NODE();
    NODE(short n, RELATION rel, ...); // Конструктор с переменным числом аргументов
};


struct FST          // Сам Конечный Автомат
{
    char* string;       // Строка, которую проверяем (например "switch")
    short position;     // Текущая позиция (какую букву читаем)
    short nstates;      // Количество состояний (узлов)
    NODE* node;         // Массив узлов
    short* rstates;     // Текущие активные состояния


    FST() { string = NULL; position = 0; nstates = 0; node = NULL; rstates = NULL; }

    // Конструктор для создания автомата из узлов
    FST(short ns, NODE n, ...);

    // Конструктор для копирования и привязки строки
    FST(char* s, FST& fst);
};
```

Рисунок 3.3 – Структура конечного автомата

Пример графа перехода конечного автомата для лексемы adv изображен на рисунке 3.4.

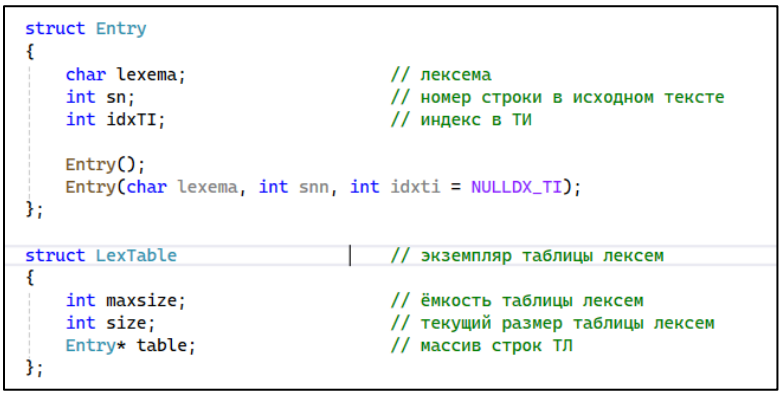
```
#define GRAPH_VAR 4, \
    FST::NODE(1,FST::RELATION('a',1)),\
    FST::NODE(1,FST::RELATION('d',2)),\
    FST::NODE(1,FST::RELATION('v',3)),\
    FST::NODE()
```

Рисунок 3.4 – Пример реализации графа КА для токена adv

Пример реализации таблицы лексем представлен в приложении Б.

3.5 Основные структуры данных

Основными структурами данных лексического анализатора являются таблица лексем и таблица идентификаторов. Таблица лексем содержит номер лексемы, лексему (lexema), полученную при разборе, номер строки в исходном коде (sn), и номер в таблице идентификаторов, если лексема является идентификаторов (idxTI). Код C++ со структурой таблицы лексем представлен на рисунке 3.5.



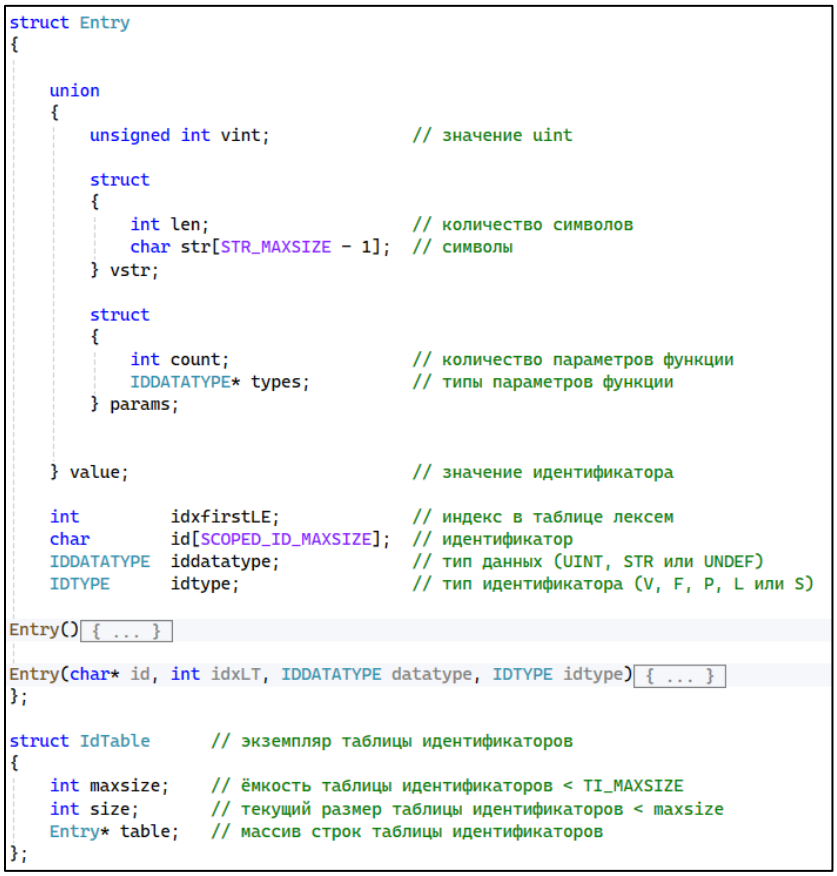
```
struct Entry
{
    char lexema;                    // лексема
    int sn;                         // номер строки в исходном тексте
    int idxTI;                      // индекс в ТИ

    Entry();
    Entry(char lexema, int snn, int idxti = NULLDX_TI);
};

struct LexTable                     // экземпляр таблицы лексем
{
    int maxsize;                    // ёмкость таблицы лексем
    int size;                       // текущий размер таблицы лексем
    Entry* table;                   // массив строк ТЛ
};
```

Рисунок 3.5 – Структура таблицы лексем

Таблица идентификаторов содержит имя идентификатора (id), номер в таблице лексем (idxfirstLE), тип данных (iddatatype), тип идентификатора (idtype) и значение (или параметры функций) (value). Код C++ со структурой таблицы идентификаторов представлен на рисунке 3.6.



```
struct Entry
{

    union
    {
        unsigned int vint;              // значение uint

        struct
        {
            int len;                    // количество символов
            char str[STR_MAXSIZE - 1];  // символы
        } vstr;

        struct
        {
            int count;                  // количество параметров функции
            IDDATATYPE* types;          // типы параметров функции
        } params;


    } value;                            // значение идентификатора

    int         idxfirstLE;             // индекс в таблице лексем
    char        id[SCOPED_ID_MAXSIZE];  // идентификатор
    IDDATATYPE  iddatatype;             // тип данных (UINT, STR или UNDEF)
    IDTYPE      idtype;                 // тип идентификатора (V, F, P, L или S)

Entry() { ... }

Entry(char* id, int idxLT, IDDATATYPE datatype, IDTYPE idtype) { ... }
};

struct IdTable      // экземпляр таблицы идентификаторов
{
    int maxsize;    // ёмкость таблицы идентификаторов < TI_MAXSIZE
    int size;       // текущий размер таблицы идентификаторов < maxsize
    Entry* table;   // массив строк таблицы идентификаторов
};
```

Рисунок 3.6 – Структура таблицы идентификатора

3.6 Структура и перечень сообщений лексического анализатора

Для обработки ошибок лексический анализатор использует централизованную таблицу сообщений. Структура сообщения включает код ошибки, уровень критичности и шаблон текста. При возникновении ошибки формируется объект исключения, содержащий номер строки и позицию в коде, где произошел сбой. Перечень сообщений об ошибках лексического анализатора (диапазон кодов 2xx) представлен на рисунке 3.7.

```
ERROR_ENTRY(200, "Лексическая ошибка: Недопустимый символ в исходном файле(-in)"),
ERROR_ENTRY(201, "Лексическая ошибка: Неизвестная последовательность символов"),
ERROR_ENTRY(202, "Лексическая ошибка: Превышен размер таблицы лексем"),
ERROR_ENTRY(203, "Лексическая ошибка: Превышен размер таблицы идентификаторов"),
ERROR_ENTRY(204, "Лексическая ошибка: Идентификатор слишком длинный (>32 символов)"),
ERROR_ENTRY(205, "Лексическая ошибка: Строковый литерал слишком длинный (>255 символов)"),
```

Рисунок 3.7 – Сообщения лексического анализатора

Такой подход с немедленной остановкой трансляции (механизм исключений) гарантирует, что на вход синтаксического анализатора не попадут заведомо некорректные данные. Благодаря фиксации точного места возникновения ошибки (номер строки и позиция символа), разработчик получает возможность быстро устранить причину сбоя, не анализируя последствия каскадных ошибок.

3.7 Принцип обработки ошибок

Ошибки, возникающие в процессе трансляции, перехватываются главным модулем программы. В случае возникновения исключительной ситуации происходит запись кода ошибки и диагностического сообщения в протокол (лог-файл), путь к которому задан входными параметрами. Используется стратегия быстрой остановки: при первой же критической ошибке трансляция прекращается.

3.8 Параметры лексического анализатора

Результаты работы анализатора сохраняются в файл журнала, задаваемый параметром -log. В консоль выводятся только общие статусы этапов выполнения. Полный вывод таблиц (ТЛ и ТИ).

3.9 Алгоритм лексического анализа

Процесс лексического анализа начинается с посимвольного считывания исходного кода и предварительной обработки входного потока. На этом этапе выполняется проверка допустимости символов и фильтрация незначащих элементов: пробелы и знаки табуляции игнорируются, однако символы перевода строки учитываются для корректной нумерации строк, что необходимо для последующей диагностики ошибок.

После очистки буфера происходит выделение очередной лексемы и попытка её распознавания с помощью цепочки детерминированных конечных автоматов (FST). В случае успешной идентификации типа токена формируется запись в

Таблице Лексем. Если распознанная лексема является идентификатором, литералом или объявлением функции, соответствующая информация дополнительно заносится в Таблицу Идентификаторов с проверкой на уникальность или повторное объявление.

Завершающим этапом алгоритма является протоколирование результатов: заполненные таблицы сохраняются в файл журнала для передачи на этап синтаксического анализа. В случае, если ни один автомат не смог распознать текущую цепочку символов, или обнаружено переполнение внутренних буферов, генерируется исключительная ситуация. Работа транслятора немедленно прекращается, а сообщение с кодом ошибки и координатами сбоя фиксируется в логе.

3.10 Контрольный пример

Корректность работы реализованного алгоритма подтверждена на контрольном примере. В ходе тестирования были успешно выделены и классифицированы все лексические единицы входного языка. Полный результат работы лексического анализатора, представленный в виде сформированных Таблицы Лексем и Таблицы Идентификаторов, приведен в приложении В.

4 Разработка синтаксического анализатора

4.1 Структура синтаксического анализатора

Синтаксический анализатор — это модуль компилятора, выполняющий проверку структуры программы на соответствие правилам формальной грамматики языка. Входной информацией для анализа являются Таблица Лексем (ТЛ) и Таблица Идентификаторов (ТИ), сформированные на предыдущем этапе. Выходной информацией является дерево разбора (цепочка примененных правил вывода) и протокол работы автомата.

Описание структуры синтаксического анализатора языка представлено на рисунке 4.1.

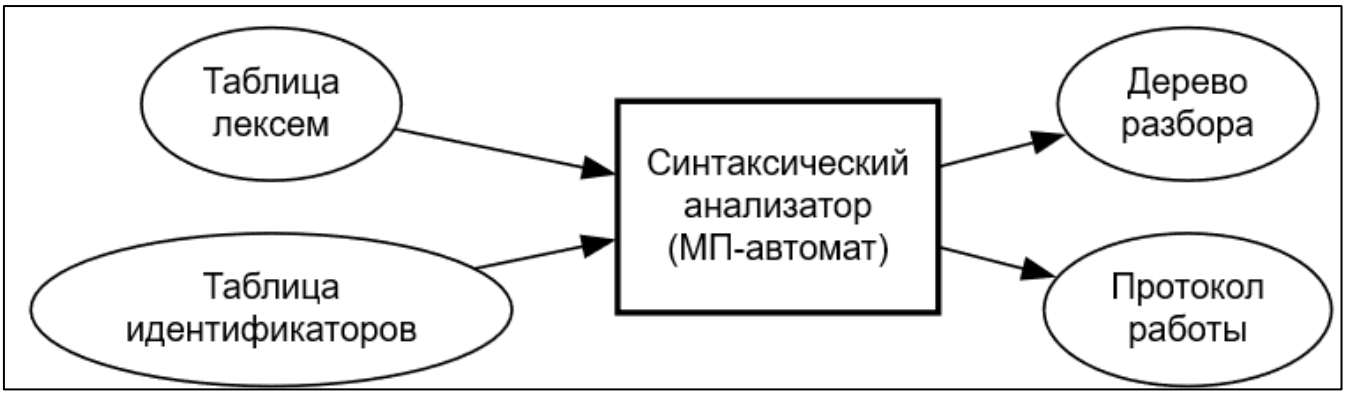


Рисунок 4.1 – Структура синтаксического анализатора

4.2 Контекстно-свободная грамматика, описывающая синтаксис языка

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

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

N – множество нетерминальных символов,

P – множество правил языка,

S – начальный символ грамматики, являющийся нетерминалом.

Для реализации нисходящего разбора грамматика приведена к нормальной форме Грейбах (НФГ). Это означает, что каждое правило начинается с терминального символа, что исключает левую рекурсию и позволяет детерминировано выбирать правило на основе очередного символа входной цепочки.

Таблица 4.1, описывающая правила грамматики языка BDV-2025 представлена в приложении Г.

4.3 Построение конечного магазинного автомата

Конечный автомат с магазинной памятью представляет собой семерку. $M = (Q, \Sigma, \Gamma, \delta, q_0, Z_0, F)$ Подробное описание компонентов магазинного автомата представлено в таблице 4.2.

Таблица 4.2 – Описание компонентов магазинного автомата

Компонент	Определение	Описание
Q	Множество состояний автомата	Состояние автомата представляет из себя структуру, содержащую позицию на входной ленте, номера текущего правила и цепочки и стек автомата
V	Алфавит входных символов	Множество терминальных символов (лексем), поступающих на вход от лексического анализатора.
Z	Алфавит специальных магазинных символов	Объединение множеств терминальных и нетерминальных символов грамматики, а также маркер дна стека.
δ	Функция переходов автомата	Функция представляет из себя множество правил грамматики, описанных в таблице 4.1.
q_0	Начальное состояние автомата	Состояние, которое приобретает автомат в начале своей работы. Представляется в виде стартового правила грамматики
z_0	Начальное состояние магазина автомата	Стартовый нетерминал S, помещаемый в магазин поверх маркера $
F	Множество конечных состояний	Конечные состояние заставляют автомат прекратить свою работу. Конечным состоянием является пустой магазин автомата и совпадение позиции на входной ленте автомата с размером ленты

Автомат использует стек для хранения ожидаемых символов. Если на вершине стека находится нетерминал, автомат заменяет его правой частью соответствующего правила. Если терминал — сравнивает с текущей лексемой входной ленты.

4.4 Основные структуры данных

Программная реализация синтаксического анализатора на языке C++ базируется на объектно-ориентированном подходе. Ключевым классом является GRB::Greibach, хранящий набор правил грамматики в нормальной форме Грейбах, где каждое правило представлено структурой GRB::Rule. Логику работы магазинного автомата реализует класс MFST::Mfst. Для поддержки алгоритма

возврата (backtracking) используется структура MFST::MfstState, которая позволяет сохранять полные снимки состояния автомата (позицию ленты и содержимое стека) на каждом шаге ветвления. Листинги данных структур представлены в приложении Д.

4.5 Описание алгоритма синтаксического разбора

В трансляторе применен алгоритм нисходящего разбора с возвратами. Процесс начинается с помещения в магазин (стек) маркера дна и стартового нетерминала S. Затем автомат считывает текущий символ из входной ленты (таблицы лексем) и анализирует вершину стека.

Если на вершине находится нетерминал, автомат ищет подходящее правило в грамматике. В случае наличия нескольких альтернатив текущее состояние сохраняется, нетерминал извлекается, а в стек помещается правая часть правила в обратном порядке. Если же на вершине стека находится терминал, он сравнивается с текущим символом на ленте. При совпадении происходит сдвиг ленты и извлечение символа из стека. При несовпадении выполняется процедура отката (backtracking) к последнему сохраненному состоянию, и выбирается альтернативное правило. Успешным завершением считается ситуация, когда стек пуст и лента прочитана полностью.

4.6 Структура и перечень сообщений синтаксического анализатора

В случае обнаружения синтаксической ошибки, когда невозможно применить ни одно правило грамматики и исчерпаны все варианты возврата, анализатор генерирует сообщение с кодом серии 6xx. Данная диагностика позволяет точно локализовать место нарушения структуры программы. Список сообщений представлен на рисунке 4.2.

```
ERROR_ENTRY(600, "Синтаксическая ошибка: Неверная структура программы (ожидается: функции или main)"),
ERROR_ENTRY(601, "Синтаксическая ошибка: Ошибка в списке параметров функции (ожидается: () или (тип имя, ...))"),
ERROR_ENTRY(602, "Синтаксическая ошибка: Ошибка в объявлении параметров (ожидается: тип имя)"),
ERROR_ENTRY(603, "Синтаксическая ошибка: Ошибка в теле программы (неверная конструкция: adv/=/cout/return/while/switch/вызов)"),
ERROR_ENTRY(604, "Синтаксическая ошибка: Ошибка в выражении (неверная арифметика, скобки или операторы)"),
ERROR_ENTRY(605, "Синтаксическая ошибка: Ошибка в списке аргументов функции (ожидается: id или literal через запятую)"),
ERROR_ENTRY(606),
ERROR_ENTRY(607),
ERROR_ENTRY(608, "Синтаксическая ошибка: Ошибка в операторе сравнения (ожидается: > < == != >= <=)"),
ERROR_ENTRY(609, "Синтаксическая ошибка: Ошибка в конструкции switch/case/default"),
```

Рисунок 4.2 – Сообщения синтаксического анализатора

Представленный набор диагностических сообщений охватывает все ключевые синтаксические конструкции языка BDV-2025, включая структуру программы, объявление функций и математические выражения. Каждому коду ошибки соответствует уникальная ситуация тупика в магазинном автомате, когда ожидаемый нетерминал не может быть раскрыт ни одним из правил грамматики для текущего входного символа. При возникновении любой из перечисленных ошибок транслятор прерывает свою работу, предотвращая генерацию некорректного ассемблерного кода и предоставляя разработчику точные координаты сбоя для исправления дефекта.

4.7 Параметры синтаксического анализатора и режимы его работы

Синтаксический анализатор запускается автоматически после успешного завершения лексического этапа. Результаты его работы, включая пошаговый протокол изменения состояния стека и последовательность примененных правил, выводятся в файл журнала (.log), путь к которому задается параметром -log. Такая детализация позволяет визуализировать процесс построения дерева вывода и точно отследить, на каком этапе произошел сбой. Режим отладки, активируемый записью в журнал, критически важен для верификации корректности грамматики.

4.8 Принцип обработки ошибок

В основу обработки ошибок положен интеллектуальный механизм диагностики, использующий структуру MfstDiagnosis. Поскольку автомат работает с возвратами, не каждое локальное несовпадение символа является ошибкой. Поэтому алгоритм отслеживает «максимально продвинутую позицию» (lenta_position) — самый глубокий индекс в таблице лексем, до которого автомату удалось добраться корректно.

Если автомат заходит в тупик, фиксируется ошибка именно в точке максимального продвижения, а не в начале конструкции. Это позволяет указать пользователю на реальное место сбоя (например, пропущенную точку с запятой внутри блока), а не на заголовок функции. Генерируется код ошибки серии 6xx, жестко привязанный к ожидаемому нетерминалу, что позволяет четко разделить структурные и содержательные ошибки. Для предотвращения зацикливания количество сохраняемых диагностических сообщений ограничено. После остановки трансляции информация преобразуется в читаемый вид и записывается в протокол.

4.9 Контрольный пример

Результаты работы синтаксического анализатора для контрольного примера приведены в приложении Е. Сформированное дерево разбора и протокол работы автомата подтверждают корректность грамматики и способность транслятора однозначно интерпретировать структуру исходного кода, соблюдая приоритеты операций и вложенность операторов.

5 Разработка семантического анализатора

5.1 Структура семантического анализатора

Семантический анализатор выполняет проверку смысловой корректности программы. Он принимает на вход результаты работы лексического и синтаксического анализаторов: Таблицу Лексем (ТЛ) и Таблицу Идентификаторов (ТИ). Этот этап является критически важным связующим звеном между парсингом и генерацией кода, так как он предотвращает создание заведомо неработоспособных программ. Без проведения семантического анализа генератор кода мог бы создать ассемблерный файл, который формально является корректным, но приводит к ошибкам времени выполнения (Runtime Errors) или неопределенному поведению программы.

В отличие от синтаксического анализатора, который проверяет структуру предложений, семантический анализатор проверяет правила использования типов данных и операций. Анализатор последовательно просматривает таблицу лексем, отслеживая контекст выполнения и валидируя совместимость операндов. В процессе работы модуль активно взаимодействует с Таблицей Идентификаторов, извлекая из нее информацию о типах переменных (UINT, STR).

Структура семантического анализатора представлена на рисунке 5.1.

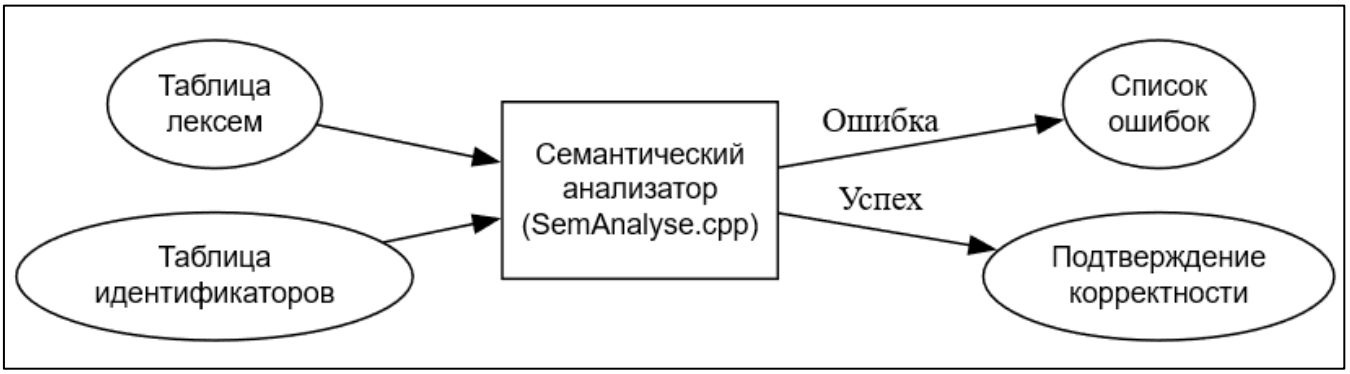


Рисунок 5.1 – Структура семантического анализатора

Схема наглядно отображает информационные связи модуля анализа с таблицами лексем и идентификаторов.

5.2 Функции семантического анализатора

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

5.3 Структура и перечень сообщений семантического анализатора

Семантический анализатор генерирует ошибки серии 3xx. Ошибки этой категории охватывают широкий спектр проблем: от несоответствия типов данных до нарушений правил вызова функций. Полный реестр диагностических сообщений

определен в файле Error.cpp, что обеспечивает централизованное управление текстами уведомлений. Перечень сообщений представлен на рисунке 5.2.

```
ERROR_ENTRY(300, "Семантическая ошибка: Необъявленный идентификатор"),
ERROR_ENTRY(301, "Семантическая ошибка: Отсутствует точка входа main"),
ERROR_ENTRY_NODEF(302),
ERROR_ENTRY_NODEF(303),
ERROR_ENTRY_NODEF(304),
ERROR_ENTRY(305, "Семантическая ошибка: Попытка переопределения идентификатора"),
ERROR_ENTRY(306, "Семантическая ошибка: Превышено максимальное количество параметров функции"),
ERROR_ENTRY(307, "Семантическая ошибка: Слишком много параметров в вызове"),
ERROR_ENTRY(308, "Семантическая ошибка: Кол-во ожидаемыех функцией и передаваемых параметров не совпадают"),
ERROR_ENTRY(309, "Семантическая ошибка: Несовпадение типов передаваемых параметров"),
ERROR_ENTRY_NODEF(310),
ERROR_ENTRY(311, "Семантическая ошибка: Обнаружен символ \'\"\'. Возможно, не закрыт строковый литерал"),
ERROR_ENTRY(312, "Семантическая ошибка: Превышен размер строкового литерала"),
ERROR_ENTRY(313, "Семантическая ошибка: Недопустимый целочисленный литерал"),
ERROR_ENTRY(314, "Семантическая ошибка: Типы данных в выражении не совпадают"),
ERROR_ENTRY(315),
ERROR_ENTRY(316, "Семантическая ошибка: Недопустимое строковое выражение справа от знака \'=\'"),
ERROR_ENTRY(317),
ERROR_ENTRY(318, "Семантическая ошибка: Деление на ноль"),
ERROR_ENTRY(319, "Семантическая ошибка: Оператор ^ требует операнды типа uint"),
ERROR_ENTRY(320, "Семантическая ошибка: Выражение в switch должно быть типа uint"),
ERROR_ENTRY(321, "Семантическая ошибка: Значение case должно быть целочисленным литералом"),
ERROR_ENTRY(322, "Семантическая ошибка: Дублирование значения case"),
ERROR_ENTRY(323, "Семантическая ошибка: switch должен содержать хотя бы один case"),
ERROR_ENTRY_NODEF(324),
ERROR_ENTRY(325, "Семантическая ошибка: Hex-литерал выходит за пределы uint"),
ERROR_ENTRY(326, "Семантическая ошибка: Операции сравнения требуют операнды одного типа"),
ERROR_ENTRY(327, "Лексическая ошибка: Ведущие нули в десятичных литералах недопустимы"),
ERROR_ENTRY(328, "Семантическая ошибка: Рекурсия не поддерживается"),
ERROR_ENTRY(329, "Семантическая ошибка: Функция должна вызываться со скобками ()"),
```

Рисунок 5.2 – Перечень сообщений семантического анализатора

Такая детализация ошибок позволяет разработчику оперативно устранять логические противоречия в коде.

5.4 Принцип обработки ошибок

При обнаружении семантической ошибки анализатор записывает сообщение в лог-файл с указанием номера строки и кода ошибки (например, ERROR 309: Несовпадение типов). В отличие от лексического и синтаксического этапов, семантический анализ пытается найти все ошибки за один проход, не останавливаясь на первой (если это возможно без нарушения логики). Это существенно ускоряет процесс отладки для пользователя. Однако генерация ассемблерного кода блокируется при наличии хотя бы одной семантической ошибки.

5.5 Контрольный пример

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

Таблица 5.1 – Примеры диагностики ошибок

Исходный код	Текст сообщения

main {
adv uint a;
adv uint a;
}
	Ошибка 305: Семантическая ошибка: Попытка переопределения идентификатора.
	Строка: 1.

main {
adv uint a;
a = 5 / 0;
}
	Ошибка 318: Семантическая ошибка: Деление на нуль.
	Строка: 3.

Как видно из приведенной таблицы, разработанный анализатор эффективно идентифицирует широкий спектр смысловых нарушений. Строгий контроль типов гарантирует целостность данных и предотвращает неявные преобразования, часто приводящие к трудноуловимым ошибкам на этапе выполнения.

6 Вычисление выражений

6.1 Выражения, допускаемые языком

В языке программирования BDV-2025 допускаются вычисления выражений, оперирующих целочисленными (uint) и строковыми (string) типами данных. Для обеспечения детерминированного порядка выполнения операций, соответствующего математическим правилам, в трансляторе реализована строгая система приоритетов. Каждой лексеме-оператору присвоен целочисленный весовой коэффициент, определяющий очередность её обработки в стеке. Наивысшим приоритетом обладают специфические для данного языка операции: унарное возведение в квадрат (обозначено символом &) и возведение в степень (символ ^). Это гарантирует, что данные вычисления будут выполнены раньше мультипликативных и аддитивных действий. Реализация этой логики находится в функции getPriority модуля PolishNotation. Полная таблица приоритетов представлена в таблице 6.1.

Таблица 6.1 – Приоритеты операций

Операция	Значение приоритета
&	5
^	4
*	3
/	3
%	3
+	2
-	2
<	1
>	1
<=	1
>=	1
==	1
!=	1
(	0
)	0

Строгое соблюдение данной иерархии гарантирует однозначность разбора выражений и служит основой для их корректного преобразования в линейную последовательность инструкций.

6.2 Польская запись и принцип её построения

Все выражения языка BDV-2025 преобразовываются в Обратную Польскую Запись (ПОЛИЗ), также известную как постфиксная нотация. Преимущество данного подхода заключается в возможности вычислять выражения за один проход, используя стековую машину, без необходимости повторного разбора скобочных структур. Это значительно упрощает этап генерации ассемблерного кода, сводя его к последовательной генерации команд push и арифметических инструкций. Алгоритм преобразования, известный как «алгоритм сортировочной станции»

Дейкстры, работает следующим образом. Входная последовательность лексем просматривается слева направо. Операнды (числа, строки, идентификаторы переменных) немедленно переносятся в выходную таблицу. Знаки операций помещаются в стек, при этом соблюдается правило приоритетов: если приоритет новой операции меньше или равен приоритету операции на вершине стека, верхний элемент выталкивается в результирующую строку. Скобки играют роль ограничителей: открывающая скобка помещается в стек, а закрывающая инициирует выталкивание всех накопленных операций до ближайшей парной открывающей скобки.

6.3 Программная реализация обработки выражений

Программная реализация алгоритма сосредоточена в пространстве имен Polish и функции PolishNotation. Данный модуль принимает на вход Таблицу Лексем и создает новую, модифицированную таблицу, где лексемы переупорядочены согласно правилам ПОЛИЗ. Процесс обработки начинается с идентификации границ выражений: алгоритм выделяет участки кода между ключевыми словами (например, после return, cout или внутри условий while) и терминаторами (точка с запятой или закрывающая скобка).

Ключевой особенностью реализации в языке BDV-2025 является гибридный подход к обработке вызовов функций. В отличие от стандартных операторов, вызовы функций требуют сохранения последовательности аргументов для корректной передачи параметров через стек. При обнаружении идентификатора функции алгоритм временно приостанавливает переупорядочивание и переносит всю конструкцию вызова, включая скобки и аргументы, в результирующую цепочку в исходном виде. Это позволяет генератору кода впоследствии корректно сформировать инструкции push для каждого аргумента и инструкцию call. Для стандартных арифметических операций используется классический стековый механизм. Особое внимание уделено операторам ^ (степень) и & (квадрат), которые обрабатываются как операции высокого приоритета, обеспечивая корректную математическую последовательность вычислений даже в сложных выражениях.

6.4 Контрольный пример

Результаты преобразования выражений контрольного примера в формат ПОЛИЗ представлены в Приложении Ж. В приложении наглядно продемонстрировано, как инфиксные выражения с учетом приоритетов операций и скобок преобразуются в линейную постфиксную последовательность. Анализ выходных данных подтверждает корректность работы алгоритма «сортировочной станции» и правильное распознавание границ выражений.

7 Генерация кода

7.1 Структура генератора кода

В языке BDV-2025 генерация кода является заключительным этапом трансляции. Генератор принимает на вход таблицы лексем и идентификаторов, полученные и обработанные на предыдущих этапах анализа. В соответствии с содержимым таблицы лексем строится выходной файл на языке ассемблера MASM, который является результатом работы транслятора. В случае обнаружения ошибок на этапах лексического, синтаксического или семантического анализа генерация кода блокируется. Структура генератора кода BDV-2025 представлена на рисунке 7.1.

Рисунок 7.1 – Структура генератора кода

7.2 Представление типов данных в оперативной памяти

Элементы таблицы идентификаторов размещаются в сегментах .data (для переменных) и .const (для литералов) языка ассемблера. Для унификации работы с памятью в 32-битной архитектуре все типы данных приводятся к 4-байтовым значениям. Тип uint хранится как двойное слово (dword), содержащее числовое значение. Тип string также хранится как dword, но содержит указатель (адрес) на начало строки в памяти. Для строковых переменных дополнительно резервируются статические буферы фиксированного размера для хранения символьных данных. Соответствие типов приведено в таблице 7.1.

Таблица 7.1 – Соответствия типов данных идентификаторов языка BDV-2025 и языка ассемблера

Тип идентификатора на языке BDV-2025	Тип идентификатора на языке ассемблера	Пояснение
uint	dword	Беззнаковое 32-бит число.
string	dword	Указатель на начало нуль-терминированной строки; литералы кладутся в .const, переменные-указатели — в .data.

7.3 Статическая библиотека

Для обеспечения работы сгенерированного ассемблерного кода используется статическая библиотека, написанная на языке C++. Эта библиотека содержит две категории функций: реализации встроенных функций языка BDV-2025 – обеспечивают выполнение функций, доступных программисту на языке BDV-2025 (описанных в разделе 1.18), сервисные функции времени выполнения – обеспечивают базовые операции ввода-вывода и управление памятью.

Таблица 7.2 – Функции статической библиотеки

Функция	Категория	Назначение
void outUint(uint value)	Сервисная	Вывод беззнакового целого числа на консоль
void outString(const char* cstr)	Сервисная	Вывод нуль-терминированной строки на консоль
unsigned int stringToUnsigned(char* buffer, const char* cstr)	Встроенная BDV-2025	Преобразование строки в число (реализация stringToUnsigned)
char* unsignedToString(unsigned int value, char* buffer)	Встроенная BDV-2025	Преобразование числа в строку (реализация unsignedToString)
unsigned int stringLength(char* buffer, const char* cstr)	Встроенная BDV-2025	Вычисление длины строки (реализация stringLength)
char* stringConcat(char* buffer, const char* left, const char* right)	Встроенная BDV-2025	Конкатенация строк (реализация stringConcat)

Объявление этих функций генерируется автоматически в ассемблерном коде. Сервисные функции outUint и outString вызываются только из сгенерированного кода для реализации оператора cout и не доступны для прямого вызова из языка BDV-2025.

7.4 Особенности алгоритма генерации кода

В языке BDV-2025 алгоритм генерации кода реализован как линейный однопроходной процесс обработки таблицы лексем. Ядром генератора является диспетчер инструкций, который определяет тип текущей лексемы и вызывает соответствующую подпрограмму генерации ассемблерного кода. Поскольку выражения предварительно преобразованы в постфиксную запись, трансляция арифметических и логических операций осуществляется по стековому принципу: операнды помещаются в стек процессора командой push, а операторы извлекают их, производят вычисления в регистрах EAX/EBX и возвращают результат. Для реализации вложенных управляющих конструкций, таких как циклы и операторы выбора, алгоритм использует вспомогательные стеки для хранения номеров меток, что позволяет корректно генерировать команды условных переходов. Блок-схема алгоритма генерации представлена на рисунке 7.2.

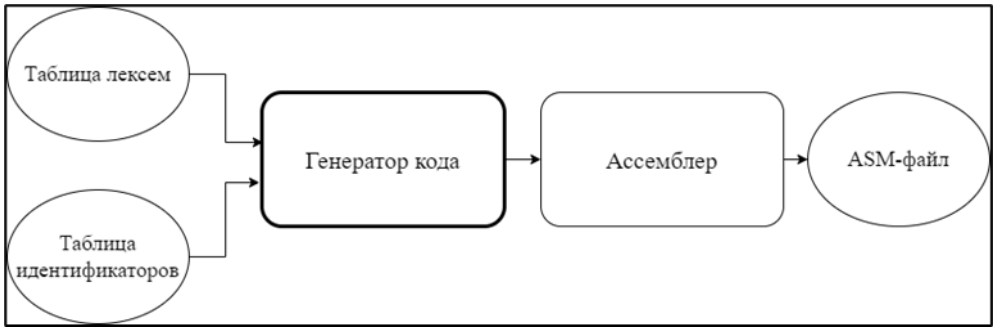


Рисунок 7.2 – Структура генератора кода

Данная схема детализирует логику работы основного цикла генератора, демонстрируя зависимость создаваемых ассемблерных инструкций от типа обрабатываемой лексемы и состояния внутренних стеков компилятора.

7.5 Входные параметры генератора кода

Входными данными для модуля являются таблицы лексем и идентификаторов, прошедшие семантический контроль, а также структура параметров запуска, определяющая пути к файлам. Генератор создает выходной поток и записывает в него директивы сегментов данных, констант и кода. Результатом работы является текстовый файл с расширением .asm, имя которого задается пользователем через параметр командной строки -out.

7.6 Контрольный пример

Результат генерации ассемблерного кода на основе контрольного примера из приложения А приведен в приложении И. Анализ листинга подтверждает корректность трансляции структур управления и математических выражений.

8 Тестирование транслятора

8.1 Общие положения

Комплексное тестирование программного продукта является неотъемлемым этапом разработки, направленным на проверку надежности транслятора и его соответствия спецификации языка BDV-2025. Процесс тестирования охватывает проверку всех этапов трансляции: лексического, синтаксического и семантического анализа, а также генерации кода. Для верификации корректности работы компилятора был разработан набор тестовых сценариев, моделирующих как корректное поведение программы, так и типичные ошибки программиста. При обнаружении несоответствий транслятор прерывает работу и формирует запись в протоколе (лог-файле), содержащую код ошибки, поясняющее диагностическое сообщение и номер строки, где произошло нарушение. Это позволяет разработчику оперативно локализовать и устранить причину сбоя.

8.2 Результаты тестирования

Таким образом, данный раздел демонстрирует способность разработанного транслятора корректно идентифицировать и классифицировать ошибки на всех этапах разбора, предоставляя пользователю информативные сообщения для отладки.

Таблица 8.1 – Тестирование лексического анализатора

Исходный код	Диагностическое сообщение
Проверка символов на допустимость	
мmain	Ошибка 200: Лексическая ошибка: Недопустимый символ в исходном файле (-in). Строка: 1
Лексический анализ	

main {
adv uint aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa;
}
	Ошибка 204: Сообщение: Лексическая ошибка: Идентификатор слишком длинный (>32 символов)

Семантический анализ	
uint function test(uint a, uint b, uint c, uint d) { return a; } main { adv uint x; x = test(1, 2, 3, 4); }	Ошибка 306: Семантическая ошибка: Превышено максимальное количество параметров функции Строка: 1
uint function factor(uint n) { return n * factor(n); } main { adv uint x; x = factor(5); }	Ошибка 328: Семантическая ошибка: Рекурсия не поддерживается Строка: 1
Синтаксический анализ	
uint function test { return 1; } main {}	Ошибка 601: строка 1, Синтаксическая ошибка: Ошибка в списке параметров функции (ожидается: () или (тип имя, ...))
main { adv uint x = 5; switch(x) { case 1 cout 1; default: cout 0;}}	Ошибка 609: строка 4, Синтаксическая ошибка: Ошибка в конструкции switch/case/default

Таким образом данный раздел предоставляет набор тестов для проверки лексического, синтаксического и семантического анализаторов.

Заключение

В ходе выполнения курсовой работы был разработан транслятор и генератор кода для языка программирования BDV-2025 со всеми необходимыми компонентами. Таким образом, были выполнены основные задачи данной курсовой работы:

Сформулирована спецификация языка BDV-2025;

– Разработаны конечные автоматы и важные алгоритмы на их основе для эффективной работы лексического анализатора;

– Осуществлена программная реализация лексического анализатора, распознающего допустимые цепочки спроектированного языка;

– Разработана контекстно-свободная, приведённая к нормальной форме Грейбаха, грамматика для описания синтаксически верных конструкций языка;

– Осуществлена программная реализация синтаксического анализатора;

– Разработан семантический анализатор, осуществляющий проверку используемых инструкций на соответствие логическим правилам;

– Разработан транслятор кода на язык ассемблера;

– Проведено тестирование всех вышеперечисленных компонентов.

Окончательная версия языка BDV-2025 включает:

– 2 типа данных;

– Поддержка оператора вывода;

– Возможность вызова 6 функций стандартной библиотеки;

– Наличие 6 арифметических операторов для вычисления выражений, наличие 6 операторов сравнения для сравнения выражений;

– Поддержка функций, операторов цикла while и оператор switch;

– Структурированная и классифицированная система для обработки ошибок пользователя.

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

Список использованных источников

1. Вирт Н. Построение компиляторов/ Пер. с англ. Борисов Е. В., Чернышов Л. Н. – М.: ДМК Пресс, 2010. – 192 с.: ил.

2. Грис Д. Конструирование компиляторов для цифровых вычислительных машин.: Пер. с англ. — М.: Мир, 1975.

3. Костельцев А. В. Построение интерпретаторов и компиляторов. СПб: Наука и Техника, 2001. — 224 стр. с ил.

4. Пратт Т. Языки программирования: разработка и реализация. Пер. с англ. — М.: Мир, 1979.

5. Хантер Р. Проектирование и конструирование компиляторов/ Пер. с англ.: Предисл. В. М. Савинкова. — М.: Финансы и статистика, 1984. — 232 с., ил.

6. Хендрикс Д. Компилятор языка Си для микроЭВМ: Пер. с англ. — М.: радио и связь, 1989. — 240 с.: ил.

Приложение А

```
uint function funcA(uint x) {
    adv uint local = x * 2;
    return local&;
}

uint function funcB(uint y) {
    adv uint local = y + 10;
    return local ^ 2003B
}

string function getNum(uint n) {
    switch(n) {
        case 1: return "one";
        case 2: return "two";
        default: return "other";
    }
}

main {
    cout "Types and literals";

    adv uint decimal = 42;
    adv uint hex = 0xFF;
    adv uint zero = 0;
    adv string text = "Hello";

    cout "decimal=42:"; cout decimal;
    cout "hex=0xFF:"; cout hex;
    cout "text:"; cout text;

    cout "";
    cout "Square & and power ^";

    adv uint a = 3;
    adv uint square = a&;
    adv uint power = a ^ 2;
    adv uint directSquare = 5&;
    adv uint directPower = 2 ^ 3;

    cout "a=3";
    cout "a& ="; cout square;
    cout "a^2 ="; cout power;
    cout "5& ="; cout directSquare;
    cout "2^3 ="; cout directPower;

    cout "";
    cout "All aritmetic operations";

    adv uint x = 20;
    adv uint y = 3;

    adv uint sum = x + y;
```

```
adv uint diff = x - y;
adv uint mul = x * y;
adv uint div = x / y;
adv uint mod = x % y;

cout "x=20, y=3";
cout "x+y ="; cout sum;
cout "x-y ="; cout diff;
cout "x*y ="; cout mul;
cout "x/y ="; cout div;
cout "x%y ="; cout mod;

cout "";
cout "Operator precedence";

adv uint prio1 = 2 + 3&;
adv uint prio2 = (2 + 3)&;
adv uint prio3 = 2 * 3&;
adv uint prio4 = 2 ^ 3 * 2;
adv uint prio5 = 10 / 2&;

cout "2+3& ="; cout prio1;
cout "(2+3)& ="; cout prio2;
cout "2*3& ="; cout prio3;
cout "2^3*2 ="; cout prio4;
cout "10/2& ="; cout prio5;

cout "";
cout "All comparison operators";

adv uint cmp1 = (5 > 3);
adv uint cmp2 = (2 == 2);
adv uint cmp3 = (4 != 5);
adv uint cmp4 = (3 < 5);
adv uint cmp5 = (5 >= 5);
adv uint cmp6 = (2 <= 3);

cout "5>3 ="; cout cmp1;
cout "2==2 ="; cout cmp2;
cout "4!=5 ="; cout cmp3;
cout "3<5 ="; cout cmp4;
cout "5>=5 ="; cout cmp5;
cout "2<=3 ="; cout cmp6;

cout "";
cout "User functions and scope";

adv uint resA = funcA(4);
adv uint resB = funcB(4);

cout "funcA(4) ="; cout resA;
cout "funcB(4) ="; cout resB;

cout "";
```

```
    cout "While loop";

    adv uint i = 1;
    while (i <= 2) {
        adv uint temp = i&;
        cout "i="; cout i;
        cout "i&="; cout temp;
        i = i + 1;
    }

    cout "";
    cout "Switch statement";

    adv uint val = 2;
    switch(val) {
        case 1: cout "ONE";
        case 2: cout "TWO";
        default: cout "DONE";
    }

    adv string word = getNum(2);
    adv string wordDefault = getNum(5);

    cout "getNum(2) ="; cout word;
    cout "getNum(5) ="; cout wordDefault;

    cout "";
    cout "All Static functions";

    adv string concat = stringConcat("Hello", "World");
    adv uint length = stringLength("Hello");
    adv uint fromStr = stringToUnsigned("123");
    adv string toStr = unsignedToString(456);

    cout "stringConcat(Hello, World) ="; cout concat;
    cout "stringLength(Hello) ="; cout length;
    cout "stringToUnsigned(123) ="; cout fromStr;
    cout "unsignedToString(456) ="; cout toStr;
}
```

Листинг 1 – Исходный код на языке BDV-2025

Приложение Б

```
// 5. adv
#define GRAPH_VAR 4, \
     FST::NODE(1,FST::RELATION('a',1)),\
     FST::NODE(1,FST::RELATION('d',2)),\
     FST::NODE(1,FST::RELATION('v',3)),\
     FST::NODE()

// 6. main
#define GRAPH_MAIN 5, \
     FST::NODE(1,FST::RELATION('m',1)),\
     FST::NODE(1,FST::RELATION('a',2)),\
     FST::NODE(1,FST::RELATION('i',3)),\
     FST::NODE(1,FST::RELATION('n',4)),\
     FST::NODE()

// 7. uint
#define GRAPH_UINT 5,\
     FST::NODE(1, FST::RELATION('u',1)),\
     FST::NODE(1, FST::RELATION('i',2)),\
     FST::NODE(1, FST::RELATION('n',3)),\
     FST::NODE(1, FST::RELATION('t',4)),\
     FST::NODE()

// 8. string
#define GRAPH_STRING 7, \
     FST::NODE(1, FST::RELATION('s',1)),\
     FST::NODE(1, FST::RELATION('t',2)),\
     FST::NODE(1, FST::RELATION('r',3)),\
     FST::NODE(1, FST::RELATION('i',4)),\
     FST::NODE(1, FST::RELATION('n',5)),\
     FST::NODE(1, FST::RELATION('g',6)),\
     FST::NODE()

// 9. function
#define GRAPH_FUNCTION 9, \
     FST::NODE(1, FST::RELATION('f', 1)),\
    FST::NODE(1, FST::RELATION('u', 2)),\
    FST::NODE(1, FST::RELATION('n', 3)),\
    FST::NODE(1, FST::RELATION('c', 4)),\
    FST::NODE(1, FST::RELATION('t', 5)),\
    FST::NODE(1, FST::RELATION('i', 6)),\
    FST::NODE(1, FST::RELATION('o', 7)),\
    FST::NODE(1, FST::RELATION('n', 8)),\
    FST::NODE()

// 10. return
#define GRAPH_RETURN 7, \
     FST::NODE(1, FST::RELATION('r',1)),\
     FST::NODE(1, FST::RELATION('e',2)),\
     FST::NODE(1, FST::RELATION('t',3)),\
     FST::NODE(1, FST::RELATION('u',4)),\
     FST::NODE(1, FST::RELATION('r',5)),\
```

```
     FST::NODE(1, FST::RELATION('n',6)),\
     FST::NODE()

// 11. cout
#define GRAPH_WRITE 5, \
     FST::NODE(1, FST::RELATION('c',1)),\
     FST::NODE(1, FST::RELATION('o',2)),\
     FST::NODE(1, FST::RELATION('u',3)),\
     FST::NODE(1, FST::RELATION('t',4)),\
     FST::NODE()

// 12. switch
#define GRAPH_SWITCH 7, \
     FST::NODE(1, FST::RELATION('s',1)),\
     FST::NODE(1, FST::RELATION('w',2)),\
     FST::NODE(1, FST::RELATION('i',3)),\
     FST::NODE(1, FST::RELATION('t',4)),\
     FST::NODE(1, FST::RELATION('c',5)),\
     FST::NODE(1, FST::RELATION('h',6)),\
     FST::NODE()

// 13. case
#define GRAPH_CASE 5, \
     FST::NODE(1, FST::RELATION('c',1)),\
     FST::NODE(1, FST::RELATION('a',2)),\
     FST::NODE(1, FST::RELATION('s',3)),\
     FST::NODE(1, FST::RELATION('e',4)),\
     FST::NODE()

// 14. default
#define GRAPH_DEFAULT 8, \
     FST::NODE(1, FST::RELATION('d',1)),\
     FST::NODE(1, FST::RELATION('e',2)),\
     FST::NODE(1, FST::RELATION('f',3)),\
     FST::NODE(1, FST::RELATION('a',4)),\
     FST::NODE(1, FST::RELATION('u',5)),\
     FST::NODE(1, FST::RELATION('l',6)),\
     FST::NODE(1, FST::RELATION('t',7)),\
     FST::NODE()

// 16. while
#define GRAPH_WHILE 6, \
     FST::NODE(1, FST::RELATION('w',1)),\
     FST::NODE(1, FST::RELATION('h',2)),\
     FST::NODE(1, FST::RELATION('i',3)),\
     FST::NODE(1, FST::RELATION('l',4)),\
     FST::NODE(1, FST::RELATION('e',5)),\
     FST::NODE()

#define GRAPH_STRTOUINT 16, \
     FST::NODE(1, FST::RELATION('s',1)),\
     FST::NODE(1, FST::RELATION('t',2)),\
     FST::NODE(1, FST::RELATION('r',3)),\
     FST::NODE(1, FST::RELATION('i',4)),\
```

```
     FST::NODE(1, FST::RELATION('n',5)),\
     FST::NODE(1, FST::RELATION('g',6)),\
     FST::NODE(1, FST::RELATION('T',7)),\
     FST::NODE(1, FST::RELATION('o',8)),\
     FST::NODE(1, FST::RELATION('U',9)),\
     FST::NODE(1, FST::RELATION('n',10)),\
     FST::NODE(1, FST::RELATION('s',11)),\
     FST::NODE(1, FST::RELATION('i',12)),\
     FST::NODE(1, FST::RELATION('g',13)),\
     FST::NODE(1, FST::RELATION('n',14)),\
     FST::NODE(1, FST::RELATION('e',15)),\
     FST::NODE(1, FST::RELATION('d',16)),\
     FST::NODE()

#define GRAPH_UINTTOSTR 16, \
     FST::NODE(1, FST::RELATION('u',1)),\
     FST::NODE(1, FST::RELATION('n',2)),\
     FST::NODE(1, FST::RELATION('s',3)),\
     FST::NODE(1, FST::RELATION('i',4)),\
     FST::NODE(1, FST::RELATION('g',5)),\
     FST::NODE(1, FST::RELATION('n',6)),\
     FST::NODE(1, FST::RELATION('e',7)),\
     FST::NODE(1, FST::RELATION('d',8)),\
     FST::NODE(1, FST::RELATION('T',9)),\
     FST::NODE(1, FST::RELATION('o',10)),\
     FST::NODE(1, FST::RELATION('S',11)),\
     FST::NODE(1, FST::RELATION('t',12)),\
     FST::NODE(1, FST::RELATION('r',13)),\
     FST::NODE(1, FST::RELATION('i',14)),\
     FST::NODE(1, FST::RELATION('n',15)),\
     FST::NODE(1, FST::RELATION('g',16)),\
     FST::NODE()

#define GRAPH_STRLEN 13, \
     FST::NODE(1, FST::RELATION('s',1)),\
     FST::NODE(1, FST::RELATION('t',2)),\
     FST::NODE(1, FST::RELATION('r',3)),\
     FST::NODE(1, FST::RELATION('i',4)),\
     FST::NODE(1, FST::RELATION('n',5)),\
     FST::NODE(1, FST::RELATION('g',6)),\
     FST::NODE(1, FST::RELATION('L',7)),\
     FST::NODE(1, FST::RELATION('e',8)),\
     FST::NODE(1, FST::RELATION('n',9)),\
     FST::NODE(1, FST::RELATION('g',10)),\
     FST::NODE(1, FST::RELATION('t',11)),\
     FST::NODE(1, FST::RELATION('h',12)),\
     FST::NODE()
#define GRAPH_STRCAT 13, \
     FST::NODE(1, FST::RELATION('s',1)),\
     FST::NODE(1, FST::RELATION('t',2)),\
     FST::NODE(1, FST::RELATION('r',3)),\
     FST::NODE(1, FST::RELATION('i',4)),\
     FST::NODE(1, FST::RELATION('n',5)),\
     FST::NODE(1, FST::RELATION('g',6)),\
```

```
    FST::NODE(1, FST::RELATION('C',7)),\
    FST::NODE(1, FST::RELATION('o',8)),\
    FST::NODE(1, FST::RELATION('n',9)),\
    FST::NODE(1, FST::RELATION('c',10)),\
    FST::NODE(1, FST::RELATION('a',11)),\
    FST::NODE(1, FST::RELATION('t',12)),\
    FST::NODE()
```

Листинг 2 — Конечные автоматы для ключевых слов языка

Приложение В

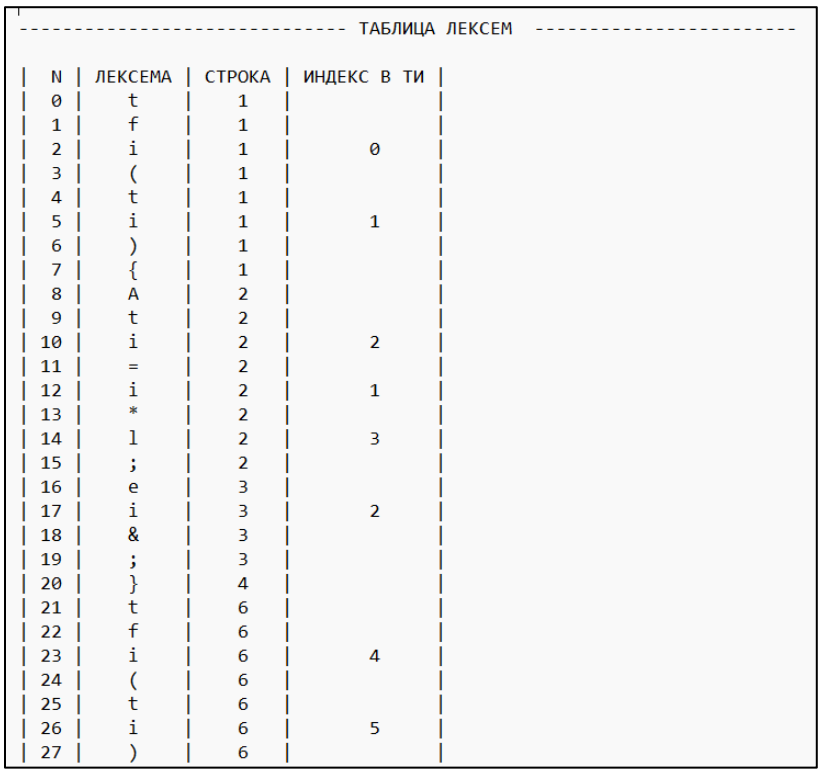
```
---------------------------  ТАБЛИЦА ИДЕНТИФИКАТОРОВ  ---------------------------

|  N  |Индекс в ЛТ| Тип идентификатора |        Тип        | Значение (параметры)
|   0 |       2   |  uint     variable |                                                  funcA |0
|   1 |       5   |  uint     variable |                                                      x |0
|   2 |      10   |  uint     variable |                                                  local |0
|   3 |      14   |  uint      literal |                                                      2 |2
|   4 |      23   |  uint     variable |                                                  funcB |0
|   5 |      26   |  uint     variable |                                                      y |0
|   6 |      35   |  uint      literal |                                                     10 |10
|   7 |      45   | string    variable |                                                 getNum |[4]etNu
|   8 |      48   |  uint     variable |                                                      n |0
|   9 |      57   |  uint      literal |                                                      1 |1
|  10 |      60   | string     literal |                                                  LIT_1 |[3]one
|  11 |      66   | string     literal |                                                  LIT_2 |[3]two
|  12 |      71   | string     literal |                                                  LIT_3 |[5]other
|  13 |      78   | string     literal |                                                  LIT_4 |[18]Types and literals
|  14 |      82   |  uint     variable |                                                decimal |0
|  15 |      84   |  uint      literal |                                                     42 |42
|  16 |      88   |  uint     variable |                                                    hex |0
|  17 |      90   |  uint      literal |                                                    255 |255
|  18 |      94   |  uint     variable |                                                   zero |0
|  19 |      96   |  uint      literal |                                                      0 |0
|  20 |     100   | string    variable |                                                   text |[2]ex
|  21 |     102   | string     literal |                                                  LIT_5 |[5]Hello
|  22 |     105   | string     literal |                                                  LIT_6 |[11]decimal=42:
|  23 |     111   | string     literal |                                                  LIT_7 |[9]hex=0xFF:
|  24 |     117   | string     literal |                                                  LIT_8 |[5]text:
|  25 |     123   | string     literal |                                                  LIT_9 |[0]
|  26 |     126   | string     literal |                                                 LIT_10 |[20]Square & and power ^
|  27 |     130   |  uint     variable |                                                      a |0
|  28 |     132   |  uint      literal |                                                      3 |3
```

Рисунок В.1 – Начало таблицы идентификаторов

```
|  93 |     553   | string     literal |                                                 LIT_48 |[16]Switch statement
|  94 |     557   |  uint     variable |                                                    val |0
|  95 |     570   | string     literal |                                                 LIT_49 |[3]ONE
|  96 |     576   | string     literal |                                                 LIT_50 |[3]TWO
|  97 |     581   | string     literal |                                                 LIT_51 |[4]DONE
|  98 |     586   | string    variable |                                                   word |[2]or
|  99 |     595   | string    variable |                                            wordDefault |[9]ordDefaul
| 100 |     603   | string     literal |                                                 LIT_52 |[11]getNum(2) =
| 101 |     609   | string     literal |                                                 LIT_53 |[11]getNum(5) =
| 102 |     615   | string     literal |                                                 LIT_54 |[0]
| 103 |     618   | string     literal |                                                 LIT_55 |[20]All Static functions
| 104 |     622   | string    variable |                                                 concat |[4]onca
| 105 |     624   | string    LIB FUNC |                                           stringConcat | P0:STRING | P1:STRING |
| 106 |     626   | string     literal |                                                 LIT_56 |[5]Hello
| 107 |     628   | string     literal |                                                 LIT_57 |[5]World
| 108 |     633   |  uint     variable |                                                 length |0
| 109 |     635   |  uint     LIB FUNC |                                           stringLength | P0:STRING |
| 110 |     637   | string     literal |                                                 LIT_58 |[5]Hello
| 111 |     642   |  uint     variable |                                                fromStr |0
| 112 |     644   |  uint     LIB FUNC |                                  stringToUnsigned | P0:STRING |
| 113 |     646   | string     literal |                                                 LIT_59 |[3]123
| 114 |     651   | string    variable |                                                  toStr |[3]oSt
| 115 |     653   | string    LIB FUNC |                                  unsignedToString | P0:UINT |
| 116 |     655   |  uint      literal |                                                    456 |456
| 117 |     659   | string     literal |                                                 LIT_60 |[28]stringConcat(Hello, World)
=
| 118 |     665   | string     literal |                                                 LIT_61 |[21]stringLength(Hello) =
| 119 |     671   | string     literal |                                                 LIT_62 |[23]stringToUnsigned(123) =
| 120 |     677   | string     literal |                                                 LIT_63 |[23]unsignedToString(456) =
```

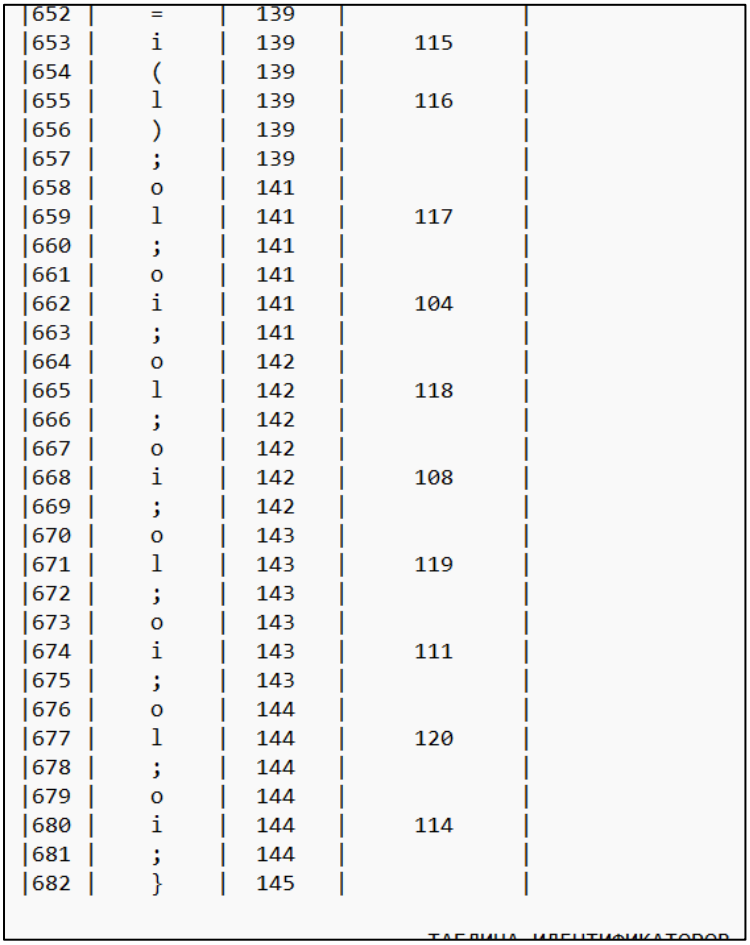
Рисунок В.2 – Конец таблицы идентификаторов



---------- ТАБЛИЦА ЛЕКСЕМ ----------

N	ЛЕКСЕМА	СТРОКА	ИНДЕКС В ТИ
0	t	1	
1	f	1	
2	i	1	0
3	(	1	
4	t	1	
5	i	1	1
6	)	1	
7	{	1	
8	A	2	
9	t	2	
10	i	2	2
11	=	2	
12	i	2	1
13	*	2	
14	l	2	3
15	;	2	
16	e	3	
17	i	3	2
18	&	3	
19	;	3	
20	}	4	
21	t	6	
22	f	6	
23	i	6	4
24	(	6	
25	t	6	
26	i	6	5
27	)	6	

Рисунок В.3 – Начало таблицы лексем



652	=	139	
653	i	139	115
654	(	139	
655	l	139	116
656	)	139	
657	;	139	
658	o	141	
659	l	141	117
660	;	141	
661	o	141	
662	i	141	104
663	;	141	
664	o	142	
665	l	142	118
666	;	142	
667	o	142	
668	i	142	108
669	;	142	
670	o	143	
671	l	143	119
672	;	143	
673	o	143	
674	i	143	111
675	;	143	
676	o	144	
677	l	144	120
678	;	144	
679	o	144	
680	i	144	114
681	;	144	
682	}	145	

Рисунок В.4 – Конец таблицы лексем

Приложение Г

Таблица 4.1 – Таблица правил переходов нетерминальных символов

Символ	Правила	Описание
S	S → t f i P { K } S S → t f i P { K } S → m { K } S S → m { K }	Стартовые правила, описывающие общую структуру программы
P	P → (E) P → ( )	Правила списка параметров функции
E	E->ti,E E->ti	Правила для параметров функции при её объявлении
K	K → A t i = W ; K K → A t i ; K K → i = W ; K K → o W ; K K → e W ; K K → W (W) { K } K K → U (W) { C } K K → i F ; K K → A t i = W ; K → A t i ; K → i = W ; K → o W ; K → e W ; K → W (W) { K } K → U (W) { C } K → i F ;	Правила для конструкций внутри функций
W	W → i W → l W → (W) W → i A W W → l A W W → i (N) W → i ( ) W → (W) A W W → i (N) A W W → i ( ) A W W → i L W W → l L W W → (W) L W W → i ^ W W → l ^ W W → (W) ^ W W → i & W → l & W → (W) & W → i & A W W → (W) & A W W → l & A W W → i & L W W → l & L W	Правила для арифметических и логических выражений

Продолжение таблицы 4.1

W	W → (W) & L W	
N	N → i N → l N → i , N N → l , N	Правила для параметров, передаваемых в функцию
F	F->(N) F->()	Правила для списка параметров, передаваемых в функцию
A	A → + A → - A → * A → / A → %	Правила, описывающие арифметические операторы
L	L → > L → < L → # L → ! L → ~ L → @	Правила, описывающие операторы сравнения
C	C → C l : K C C → C l : K C → D : K C C → D : K	Правило, описывающее тело функции/условного выражения

Приложение Д

```
namespace GRB
{
     struct Rule    //правило в грамматике Грейбах
     {
          GRBALPHABET  nn;    //нетерминал(левый символ правила) <0
          int iderror;        //идентификатор диагностического
сообщения
          short size;              //количество цепочек - правых
частей правила

          struct Chain        //цепочка(правая часть правила)
          {
               short size;                              //длина
цепочки
               GRBALPHABET* nt;                         //цепочка
терминалов(>0 и нетерминалов (<0)
               Chain() { size = 0; nt = nullptr; };
               Chain(
                    short psize,                  //количество
символов в цепочке
                    GRBALPHABET s, ...            //символы (терминал
или нетерминал)
               );
               char* getCChain(char* b);          //получить правую
сторону правила
               static GRBALPHABET T(char t) { return GRBALPHABET(t);
};//терминал
               static GRBALPHABET N(char n) { return -GRBALPHABET(n);
};//нетерминал
               static bool isT(GRBALPHABET s) { return s > 0; }
          //терминал?
               static bool isN(GRBALPHABET s) { return !isT(s); }
          //нетерминал?
               static char alphabet_to_char(GRBALPHABET s) { return
isT(s) ? char(s) : char(-s); };//GRBALPHABET->char
          }*chains; //массив цепочек - правых частей правила

          Rule() { nn = 0x00; size = 0; iderror = 0; chains =
nullptr; }
          Rule(
               GRBALPHABET pnn,              //нетерминал (<0)
               int iderror,                  //идентификатор
диагностического сообщения
               short psize,                  //количество цепочек -
правых частей правила
               Chain c, ...                  //множество цепочек -
правых частей правила
          );
          char* getCRule(                    //получить правило в
виде N->цепочка(для распечатки)
               char* b,                      //буфер
```

```
                short nchain              //номер цепочки(правой части)
в правиле
          );
          short getNextChain( //получить следующую за j подходящую
цепочку, вернуть её номер или -1
                GRBALPHABET t,            //первый символ цепочки
                Rule::Chain& pchain,      //возвращаемая цепочка
                short j                        //номер цепочки
          );
     };

     struct Greibach                //грамматика Грейбах
     {
          short size;               //количество правил
          GRBALPHABET startN; //стартовый символ
          GRBALPHABET stbottomT;//дно стека
          Rule* rules;              //множество правил
          Greibach() { size = 0; startN = 0; stbottomT = 0; rules =
nullptr; };
          Greibach(
                GRBALPHABET pstartN,          //стартовый символ
                GRBALPHABET pstbootomT,       //дно стека
                short psize,                  //количество правил
                Rule r, ...                        //правила
          );
          short getRule(      //получить правило, возвращается номер
правила или -1
                GRBALPHABET pnn,    //левый символ правила
                Rule& prule               //возвращаемое правило
грамматики
          );
          Rule getRule(short n);    //получить правило по номеру
     };
     Greibach getGreibach();        //получить грамматику
};
```

Листинг 3 – Структура грамматики Грейбах

```
namespace MFST
{
     struct MfstState                  //состояние автомата(для
сохранения
     {
          short lenta_position;         //позиция на ленте
          short nrule;                  //номер текущего правила
          short nrulechain;             //номер текущей цепочки
          MFSTSTSTACK st;                    //стек автомата
          MfstState();
          MfstState(
                short pposition,         //позиция на ленте
                MFSTSTSTACK pst,         //стек автомата
                short pnrulechain        //номер текущей цепочки,
текущего правила
          );
```

```
          MfstState(
               short pposition,          //позиция на ленте
               MFSTSTSTACK pst,          //стек автомата
               short pnrule,             //номер текущего правила
               short pnrulechain         //номер текущей цепочки,
текущего правила
          );
     };

     struct Mfst                         //магазинный автомат
     {
          enum RC_STEP              //шаг автомата
          {
               NS_OK,                    //найдено правило и цепочка,
цепочка записана в стек
               NS_NORULE,                //не найдено правило
грамматики (ошибки в грамматике)
               NS_NORULECHAIN,           //не найдена подходящая
цепочка правила (ошибка в исходном коде)
               NS_ERROR,            //неизвестный нетерминальный
символ грамматики
               TS_OK,                    //текущий символ ленты ==
вершине стека, продвинулась лента, pop стека
               TS_NOK,                   //текущий символ ленты !=
вершине стека, восстановлено состояние
               LENTA_END,                //текущая позиция ленты >=
lenta_size
               SURPRISE             //неожиданный код возврата (
ошибка в step)
          };
          struct MfstDiagnosis           //диагностика
          {
               short lenta_position;     //позиция на ленте
               RC_STEP rc_step;          //код завершения шага
               short nrule;              //номер правила
               short nrule_chain;        //номер цепочки правила
               MfstDiagnosis();
               MfstDiagnosis(            //диагностика
                    short plenta_position,    //позиция на ленте
                    RC_STEP prc_step,         //код завершения шага
                    short pnrule,             //номер правила
                    short pnrule_chain        //номер цепочки правила
               );
          }
          diagnosis[MFST_DIAGN_digit];  //последние самые глубокие
сообщения
          class my_stack_MfstState :public std::stack<MfstState> {
          public:
               using std::stack<MfstState>::c;
          };

          GRBALPHABET* lenta;
     //перекодированная(TS/NS) лента (из LEX)
```

```
          short lenta_position;              //текущая позиция на
ленте
          short nrule;                       //номер текущего правила
          short nrulechain;                  //номер текущей цепочки,
текущего правила
          short lenta_size;                  //размер ленты
          GRB::Greibach grebach;             //грамматика Грейбах
          Lexer::LEX lex;                         //результат работы
лексического анализатора
          MFSTSTSTACK st;                         //стек автомата

          my_stack_MfstState storestate;     //стек для сохранения
состояний
          Mfst();
          Mfst(
               Lexer::LEX plex,              //результат работы
лексического анализатора
               GRB::Greibach pgrebach        //грамматика Грейбах
          );
          char* getCSt(char* buf);           //получить содержимое
стека
          char* getCLenta(char* buf, short pos, short n =
25);//лента: n символов с pos
          char* getDiagnosis(short n, char* buf);
     //получить n-ую строку диагностики или 0x00
          bool savestate(const Log::LOG& log);
     //сохранить состояние автомата
          bool reststate(const Log::LOG& log);
     //восстановить состояние автомата
          bool push_chain(                        //поместить уепочку
правила в стек
               GRB::Rule::Chain chain                  //цепочка
правила
          );
          RC_STEP step(const Log::LOG& log);           //выполнить
шаг автомата
          bool start(const Log::LOG& log);             //запустить
автомат
          bool savediagnois(
               RC_STEP pprc_step             //код завершения шага
          );
          void printrules(const Log::LOG& log);        //вывести
последовательность правил
          struct Deducation        //вывод
          {
               short size;         //количество шагов в выводе
               short* nrules; //номера правил грамматики
               short* nrulechains;//номера цепочек правил грамматики
(nrules)
               Deducation() { size = 0; nrules = 0; nrulechains = 0;
};
} deducation;bool savededucation();     //сохранить дерево вывода};};
```

Листинг 4 – Структура магазинного конечного автомата

Приложение Е

```
0   : S->tfiP{K}S
3   : P->(E)
4   : E->ti
8   : K->Ati=W;K
12  : W->iAW
13  : A->*
14  : W->l
16  : K->eW;
17  : W->i&
21  : S->tfiP{K}S
24  : P->(E)
25  : E->ti
29  : K->Ati=W;K
33  : W->iAW
34  : A->+
35  : W->l
37  : K->eW;
38  : W->i^W
40  : W->l
43  : S->tfiP{K}S
46  : P->(E)
47  : E->ti
51  : K->U(W){C}
53  : W->i
56  : C->Cl:KC
59  : K->eW;
60  : W->l
62  : C->Cl:KC
65  : K->eW;
66  : W->l
68  : C->D:K
```

Рисунок Е.1 – Начало дерева разбора

```
617 : K->oW;K
618 : W->l
620 : K->Ati=W;K
624 : W->i(N)
626 : N->l,N
628 : N->l
631 : K->Ati=W;K
635 : W->i(N)
637 : N->l
640 : K->Ati=W;K
644 : W->i(N)
646 : N->l
649 : K->Ati=W;K
653 : W->i(N)
655 : N->l
658 : K->oW;K
659 : W->l
661 : K->oW;K
662 : W->i
664 : K->oW;K
665 : W->l
667 : K->oW;K
668 : W->i
670 : K->oW;K
671 : W->l
673 : K->oW;K
674 : W->i
676 : K->oW;K
677 : W->l
679 : K->oW;
680 : W->i
```

Рисунок Е.2 – Конец дерева разбора

Приложение Ж

```
 1 | tfi[0](ti[1]){
 2 | Ati[2]=i[1]*l[3];
 3 | ei[2]&;
 4 | }
 6 | tfi[4](ti[5]){
 7 | Ati[2]=i[5]+l[6];
 8 | ei[2]^l[3];
 9 | }
11 | tfi[7](ti[8]){
12 | U(i[8]){
13 | Cl[9]:el[10];
14 | Cl[3]:el[11];
15 | D:el[12];
16 | }
17 | }
19 | m{
20 | ol[13];
22 | Ati[14]=l[15];
23 | Ati[16]=l[17];
24 | Ati[18]=l[19];
25 | Ati[20]=l[21];
27 | ol[22];oi[14];
28 | ol[23];oi[16];
29 | ol[24];oi[20];
31 | ol[25];
32 | ol[26];
34 | Ati[27]=l[28];
35 | Ati[29]=i[27]&;
36 | Ati[30]=i[27]^l[3];
```

Рисунок Ж.1 – Промежуточное представление кода (начало)

```
113 | ol[91];oi[89];
114 | i[88]=i[88]+l[9];
115 | }
117 | ol[92];
118 | ol[93];
120 | Ati[94]=l[3];
121 | U(i[94]){
122 | Cl[9]:ol[95];
123 | Cl[3]:ol[96];
124 | D:ol[97];
125 | }
127 | Ati[98]=i[7](l[3]);
128 | Ati[99]=i[7](l[32]);
130 | ol[100];oi[98];
131 | ol[101];oi[99];
133 | ol[102];
134 | ol[103];
136 | Ati[104]=i[105](l[106],l[107]);
137 | Ati[108]=i[109](l[110]);
138 | Ati[111]=i[112](l[113]);
139 | Ati[114]=i[115](l[116]);
141 | ol[117];oi[104];
142 | ol[118];oi[108];
143 | ol[119];oi[111];
144 | ol[120];oi[114];
145 | }
```

Рисунок Ж.2 – Промежуточное представление кода (конец)

Приложение И

```
.586
.model flat, stdcall
includelib libucrt.lib
includelib kernel32.lib
includelib "..\Debug\StaticLibrary.lib"
ExitProcess PROTO :DWORD
.stack 4096

 stringToUnsigned PROTO : DWORD, : DWORD

 unsignedToString PROTO : DWORD, : DWORD

 stringLength PROTO : DWORD, : DWORD

 stringConcat PROTO : DWORD, : DWORD, : DWORD

 outUint PROTO : DWORD

 outString PROTO : DWORD
.const
          newline byte 13, 10, 0
.data
          temp sdword ?
          buffer byte 256 dup(0)
          temp_str_buf byte 256 dup(0)
          v_x dword ?
          v_local dword ?
          L_2 dword 2
          v_y dword ?
          L_10 dword 10
          v_n dword ?
          L_1 dword 1
          L_LIT_1 byte "one", 0
          L_LIT_2 byte "two", 0
          L_LIT_3 byte "other", 0
          L_LIT_4 byte "Types and literals", 0
          v_decimal dword ?
          L_42 dword 42
          v_hex dword ?
          L_255 dword 255
          v_zero dword ?
          L_0 dword 0
          v_text dword ?
          v_text_buf byte 256 dup(0)
          L_LIT_5 byte "Hello", 0
          L_LIT_6 byte "decimal=42:", 0
          L_LIT_7 byte "hex=0xFF:", 0
          L_LIT_8 byte "text:", 0
          L_LIT_9 byte 0
          L_LIT_10 byte "Square & and power ^", 0
          v_a dword ?
          L_3 dword 3
```

```
v_square dword ?
v_power dword ?
v_directSquare dword ?
L_5 dword 5
v_directPower dword ?
L_LIT_11 byte "a=3", 0
L_LIT_12 byte "a& =", 0
L_LIT_13 byte "a^2 =", 0
L_LIT_14 byte "5& =", 0
L_LIT_15 byte "2^3 =", 0
L_LIT_16 byte 0
L_LIT_17 byte "All aritmetic operations", 0
L_20 dword 20
v_sum dword ?
v_diff dword ?
v_mul dword ?
v_div dword ?
v_mod dword ?
L_LIT_18 byte "x=20, y=3", 0
L_LIT_19 byte "x+y =", 0
L_LIT_20 byte "x-y =", 0
L_LIT_21 byte "x*y =", 0
L_LIT_22 byte "x/y =", 0
L_LIT_23 byte "x%y =", 0
L_LIT_24 byte 0
L_LIT_25 byte "Operator precedence", 0
v_prio1 dword ?
v_prio2 dword ?
v_prio3 dword ?
v_prio4 dword ?
v_prio5 dword ?
L_LIT_26 byte "2+3& =", 0
L_LIT_27 byte "(2+3)& =", 0
L_LIT_28 byte "2*3& =", 0
L_LIT_29 byte "2^3*2 =", 0
L_LIT_30 byte "10/2& =", 0
L_LIT_31 byte 0
L_LIT_32 byte "All comparison operators", 0
v_cmp1 dword ?
v_cmp2 dword ?
v_cmp3 dword ?
L_4 dword 4
v_cmp4 dword ?
v_cmp5 dword ?
v_cmp6 dword ?
L_LIT_33 byte "5>3 =", 0
L_LIT_34 byte "2==2 =", 0
L_LIT_35 byte "4!=5 =", 0
L_LIT_36 byte "3<5 =", 0
L_LIT_37 byte "5>=5 =", 0
L_LIT_38 byte "2<=3 =", 0
L_LIT_39 byte 0
L_LIT_40 byte "User functions and scope", 0
v_resA dword ?
```

```
            v_resB dword ?
            L_LIT_41 byte "funcA(4) =", 0
            L_LIT_42 byte "funcB(4) =", 0
            L_LIT_43 byte 0
            L_LIT_44 byte "While loop", 0
            v_i dword ?
            v_temp dword ?
            L_LIT_45 byte "i=", 0
            L_LIT_46 byte "i&=", 0
            L_LIT_47 byte 0
            L_LIT_48 byte "Switch statement", 0
            v_val dword ?
            L_LIT_49 byte "ONE", 0
            L_LIT_50 byte "TWO", 0
            L_LIT_51 byte "DONE", 0
            v_word dword ?
            v_word_buf byte 256 dup(0)
            v_wordDefault dword ?
            v_wordDefault_buf byte 256 dup(0)
            L_LIT_52 byte "getNum(2) =", 0
            L_LIT_53 byte "getNum(5) =", 0
            L_LIT_54 byte 0
            L_LIT_55 byte "All Static functions", 0
            v_concat dword ?
            v_concat_buf byte 256 dup(0)
            L_LIT_56 byte "Hello", 0
            L_LIT_57 byte "World", 0
            v_length dword ?
            L_LIT_58 byte "Hello", 0
            v_fromStr dword ?
            L_LIT_59 byte "123", 0
            v_toStr dword ?
            v_toStr_buf byte 256 dup(0)
            L_456 dword 456
            L_LIT_60 byte "stringConcat(Hello, World) =", 0
            L_LIT_61 byte "stringLength(Hello) =", 0
            L_LIT_62 byte "stringToUnsigned(123) =", 0
            L_LIT_63 byte "unsignedToString(456) =", 0
.code
;----------- fn_funcA ------------
fn_funcA PROC p_x : dword
; --- save registers ---
push ebx
push edx
; ----------------------
push p_x
push L_2
pop ebx
pop eax
imul eax, ebx
push eax
pop eax
mov v_local, eax
push v_local
```

```
pop eax
imul eax, eax
push eax
pop eax
; --- restore registers ---
pop edx
pop ebx
; ------------------------
ret 4
fn_funcA ENDP
;-----------------------------

;----------- fn_funcB ------------
fn_funcB PROC p_y : dword
; --- save registers ---
push ebx
push edx
; ----------------------
push p_y
push L_10
pop ebx
pop eax
add eax, ebx
push eax
pop eax
mov v_local, eax
push v_local
push L_2
pop ecx
pop eax
mov ebx, eax
cmp ecx, 0
jne pow_1_start
mov eax, 1
jmp pow_1_end
pow_1_start:
dec ecx
pow_1_loop:
cmp ecx, 0
je pow_1_end
imul eax, ebx
dec ecx
jmp pow_1_loop
pow_1_end:
push eax
pop eax
; --- restore registers ---
pop edx
pop ebx
; ------------------------
ret 4
fn_funcB ENDP
;-----------------------------
```

```
;----------- fn_getNum ------------
fn_getNum PROC p_n : dword
; --- save registers ---
push ebx
push edx
; ----------------------
; --- switch ---
mov eax, p_n
cmp eax, 1
je case_1_0
cmp eax, 2
je case_1_1
jmp default_1
case_1_0:
push offset L_LIT_1
pop eax
; --- restore registers ---
pop edx
pop ebx
; -------------------------
ret 4
jmp switch_1_end
case_1_1:
push offset L_LIT_2
pop eax
; --- restore registers ---
pop edx
pop ebx
; -------------------------
ret 4
jmp switch_1_end
default_1:
push offset L_LIT_3
pop eax
; --- restore registers ---
pop edx
pop ebx
; -------------------------
ret 4
switch_1_end:
fn_getNum ENDP
;------------------------------

;----------- MAIN ------------
main PROC
push offset L_LIT_4
call outString
push offset newline
call outString
push L_42
pop eax
mov v_decimal, eax
push L_255
pop eax
```

```
mov v_hex, eax
push L_0
pop eax
mov v_zero, eax
mov v_text, offset L_LIT_5
push offset L_LIT_6
call outString
push offset newline
call outString
push v_decimal
call outUint
push offset newline
call outString
push offset L_LIT_7
call outString
push offset newline
call outString
push v_hex
call outUint
push offset newline
call outString
push offset L_LIT_8
call outString
push offset newline
call outString
push v_text
call outString
push offset newline
call outString
push offset L_LIT_9
call outString
push offset newline
call outString
push offset L_LIT_10
call outString
push offset newline
call outString
push L_3
pop eax
mov v_a, eax
push v_a
pop eax
imul eax, eax
push eax
pop eax
mov v_square, eax
push v_a
push L_2
pop ecx
pop eax
mov ebx, eax
cmp ecx, 0
jne pow_2_start
mov eax, 1
```

```
jmp pow_2_end
pow_2_start:
dec ecx
pow_2_loop:
cmp ecx, 0
je pow_2_end
imul eax, ebx
dec ecx
jmp pow_2_loop
pow_2_end:
push eax
pop eax
mov v_power, eax
push L_5
pop eax
imul eax, eax
push eax
pop eax
mov v_directSquare, eax
push L_2
push L_3
pop ecx
pop eax
mov ebx, eax
cmp ecx, 0
jne pow_3_start
mov eax, 1
jmp pow_3_end
pow_3_start:
dec ecx
pow_3_loop:
cmp ecx, 0
je pow_3_end
imul eax, ebx
dec ecx
jmp pow_3_loop
pow_3_end:
push eax
pop eax
mov v_directPower, eax
push offset L_LIT_11
call outString
push offset newline
call outString
push offset L_LIT_12
call outString
push offset newline
call outString
push v_square
call outUint
push offset newline
call outString
push offset L_LIT_13
call outString
```

```
push offset newline
call outString
push v_power
call outUint
push offset newline
call outString
push offset L_LIT_14
call outString
push offset newline
call outString
push v_directSquare
call outUint
push offset newline
call outString
push offset L_LIT_15
call outString
push offset newline
call outString
push v_directPower
call outUint
push offset newline
call outString
push offset L_LIT_16
call outString
push offset newline
call outString
push offset L_LIT_17
call outString
push offset newline
call outString
push L_20
pop eax
mov v_x, eax
push L_3
pop eax
mov v_y, eax
push v_x
push v_y
pop ebx
pop eax
add eax, ebx
push eax
pop eax
mov v_sum, eax
push v_x
push v_y
pop ebx
pop eax
sub eax, ebx
push eax
pop eax
mov v_diff, eax
push v_x
push v_y
```

```
pop ebx
pop eax
imul eax, ebx
push eax
pop eax
mov v_mul, eax
push v_x
push v_y
pop ebx
pop eax
cmp ebx, 0
je divsafe_4_zero
xor edx, edx
div ebx
push eax
jmp divsafe_4_end
divsafe_4_zero:
push 0
divsafe_4_end:
pop eax
mov v_div, eax
push v_x
push v_y
pop ebx
pop eax
cmp ebx, 0
je modsafe_5_zero
xor edx, edx
div ebx
push edx
jmp modsafe_5_end
modsafe_5_zero:
push 0
modsafe_5_end:
pop eax
mov v_mod, eax
push offset L_LIT_18
call outString
push offset newline
call outString
push offset L_LIT_19
call outString
push offset newline
call outString
push v_sum
call outUint
push offset newline
call outString
push offset L_LIT_20
call outString
push offset newline
call outString
push v_diff
call outUint
```

```
push offset newline
call outString
push offset L_LIT_21
call outString
push offset newline
call outString
push v_mul
call outUint
push offset newline
call outString
push offset L_LIT_22
call outString
push offset newline
call outString
push v_div
call outUint
push offset newline
call outString
push offset L_LIT_23
call outString
push offset newline
call outString
push v_mod
call outUint
push offset newline
call outString
push offset L_LIT_24
call outString
push offset newline
call outString
push offset L_LIT_25
call outString
push offset newline
call outString
push L_2
push L_3
pop eax
imul eax, eax
push eax
pop ebx
pop eax
add eax, ebx
push eax
pop eax
mov v_prio1, eax
push L_2
push L_3
pop ebx
pop eax
add eax, ebx
push eax
pop eax
imul eax, eax
push eax
```

```
pop eax
mov v_prio2, eax
push L_2
push L_3
pop eax
imul eax, eax
push eax
pop ebx
pop eax
imul eax, ebx
push eax
pop eax
mov v_prio3, eax
push L_2
push L_3
pop ecx
pop eax
mov ebx, eax
cmp ecx, 0
jne pow_6_start
mov eax, 1
jmp pow_6_end
pow_6_start:
dec ecx
pow_6_loop:
cmp ecx, 0
je pow_6_end
imul eax, ebx
dec ecx
jmp pow_6_loop
pow_6_end:
push eax
push L_2
pop ebx
pop eax
imul eax, ebx
push eax
pop eax
mov v_prio4, eax
push L_10
push L_2
pop eax
imul eax, eax
push eax
pop ebx
pop eax
cmp ebx, 0
je divsafe_7_zero
xor edx, edx
div ebx
push eax
jmp divsafe_7_end
divsafe_7_zero:
push 0
```

```
divsafe_7_end:
pop eax
mov v_prio5, eax
push offset L_LIT_26
call outString
push offset newline
call outString
push v_prio1
call outUint
push offset newline
call outString
push offset L_LIT_27
call outString
push offset newline
call outString
push v_prio2
call outUint
push offset newline
call outString
push offset L_LIT_28
call outString
push offset newline
call outString
push v_prio3
call outUint
push offset newline
call outString
push offset L_LIT_29
call outString
push offset newline
call outString
push v_prio4
call outUint
push offset newline
call outString
push offset L_LIT_30
call outString
push offset newline
call outString
push v_prio5
call outUint
push offset newline
call outString
push offset L_LIT_31
call outString
push offset newline
call outString
push offset L_LIT_32
call outString
push offset newline
call outString
push L_5
push L_3
pop ebx
```

```
pop eax
cmp eax, ebx
jg cmp_true_kpppdmpd
mov eax, 0
jmp cmp_end_kpppdmpd
cmp_true_kpppdmpd:
mov eax, 1
cmp_end_kpppdmpd:
push eax
pop eax
mov v_cmp1, eax
push L_2
push L_2
pop ebx
pop eax
cmp eax, ebx
je cmp_true_kpppdmpdc
mov eax, 0
jmp cmp_end_kpppdmpdc
cmp_true_kpppdmpdc:
mov eax, 1
cmp_end_kpppdmpdc:
push eax
pop eax
mov v_cmp2, eax
push L_4
push L_5
pop ebx
pop eax
cmp eax, ebx
jne cmp_true_kpppdmpdcc
mov eax, 0
jmp cmp_end_kpppdmpdcc
cmp_true_kpppdmpdcc:
mov eax, 1
cmp_end_kpppdmpdcc:
push eax
pop eax
mov v_cmp3, eax
push L_3
push L_5
pop ebx
pop eax
cmp eax, ebx
jl cmp_true_kpppdmpdccc
mov eax, 0
jmp cmp_end_kpppdmpdccc
cmp_true_kpppdmpdccc:
mov eax, 1
cmp_end_kpppdmpdccc:
push eax
pop eax
mov v_cmp4, eax
push L_5
```

```
push L_5
pop ebx
pop eax
cmp eax, ebx
jge cmp_true_kpppdmpdcccc
mov eax, 0
jmp cmp_end_kpppdmpdcccc
cmp_true_kpppdmpdcccc:
mov eax, 1
cmp_end_kpppdmpdcccc:
push eax
pop eax
mov v_cmp5, eax
push L_2
push L_3
pop ebx
pop eax
cmp eax, ebx
jle cmp_true_kpppdmpdccccc
mov eax, 0
jmp cmp_end_kpppdmpdccccc
cmp_true_kpppdmpdccccc:
mov eax, 1
cmp_end_kpppdmpdccccc:
push eax
pop eax
mov v_cmp6, eax
push offset L_LIT_33
call outString
push offset newline
call outString
push v_cmp1
call outUint
push offset newline
call outString
push offset L_LIT_34
call outString
push offset newline
call outString
push v_cmp2
call outUint
push offset newline
call outString
push offset L_LIT_35
call outString
push offset newline
call outString
push v_cmp3
call outUint
push offset newline
call outString
push offset L_LIT_36
call outString
push offset newline
```

```
call outString
push v_cmp4
call outUint
push offset newline
call outString
push offset L_LIT_37
call outString
push offset newline
call outString
push v_cmp5
call outUint
push offset newline
call outString
push offset L_LIT_38
call outString
push offset newline
call outString
push v_cmp6
call outUint
push offset newline
call outString
push offset L_LIT_39
call outString
push offset newline
call outString
push offset L_LIT_40
call outString
push offset newline
call outString

push L_4
call fn_funcA
push eax
pop eax
mov v_resA, eax

push L_4
call fn_funcB
push eax
pop eax
mov v_resB, eax
push offset L_LIT_41
call outString
push offset newline
call outString
push v_resA
call outUint
push offset newline
call outString
push offset L_LIT_42
call outString
push offset newline
call outString
push v_resB
```

```
call outUint
push offset newline
call outString
push offset L_LIT_43
call outString
push offset newline
call outString
push offset L_LIT_44
call outString
push offset newline
call outString
push L_1
pop eax
mov v_i, eax
while_1_start:
push v_i
push L_2
pop ebx
pop eax
cmp eax, ebx
jle cmp_true_kpppdmpdcccccc
mov eax, 0
jmp cmp_end_kpppdmpdcccccc
cmp_true_kpppdmpdcccccc:
mov eax, 1
cmp_end_kpppdmpdcccccc:
push eax
pop eax
cmp eax, 0
je while_1_end
push v_i
pop eax
imul eax, eax
push eax
pop eax
mov v_temp, eax
push offset L_LIT_45
call outString
push offset newline
call outString
push v_i
call outUint
push offset newline
call outString
push offset L_LIT_46
call outString
push offset newline
call outString
push v_temp
call outUint
push offset newline
call outString
push v_i
push L_1
```

```
pop ebx
pop eax
add eax, ebx
push eax
pop eax
mov v_i, eax
jmp while_1_start
while_1_end:
push offset L_LIT_47
call outString
push offset newline
call outString
push offset L_LIT_48
call outString
push offset newline
call outString
push L_2
pop eax
mov v_val, eax
; --- switch ---
mov eax, v_val
cmp eax, 1
je case_2_0
cmp eax, 2
je case_2_1
jmp default_2
case_2_0:
push offset L_LIT_49
call outString
push offset newline
call outString
jmp switch_2_end
case_2_1:
push offset L_LIT_50
call outString
push offset newline
call outString
jmp switch_2_end
default_2:
push offset L_LIT_51
call outString
push offset newline
call outString
switch_2_end:

push L_2
call fn_getNum
mov v_word, eax

push L_5
call fn_getNum
mov v_wordDefault, eax
push offset L_LIT_52
call outString
```

```
push offset newline
call outString
push v_word
call outString
push offset newline
call outString
push offset L_LIT_53
call outString
push offset newline
call outString
push v_wordDefault
call outString
push offset newline
call outString
push offset L_LIT_54
call outString
push offset newline
call outString
push offset L_LIT_55
call outString
push offset newline
call outString

push offset L_LIT_57
push offset L_LIT_56
push offset buffer
call stringConcat
; Copy string from buffer to variable's buffer
push esi
push edi
mov esi, eax
mov edi, offset v_concat_buf
copy_15_loop:
mov al, [esi]
mov [edi], al
cmp al, 0
je copy_15_end
inc esi
inc edi
jmp copy_15_loop
copy_15_end:
pop edi
pop esi
mov v_concat, offset v_concat_buf

push offset L_LIT_58
push offset buffer
call stringLength
push eax
pop eax
mov v_length, eax

push offset L_LIT_59
push offset buffer
```

```
call stringToUnsigned
push eax
pop eax
mov v_fromStr, eax

push L_456
push offset buffer
call unsignedToString
; Copy string from buffer to variable's buffer
push esi
push edi
mov esi, eax
mov edi, offset v_toStr_buf
copy_16_loop:
mov al, [esi]
mov [edi], al
cmp al, 0
je copy_16_end
inc esi
inc edi
jmp copy_16_loop
copy_16_end:
pop edi
pop esi
mov v_toStr, offset v_toStr_buf
push offset L_LIT_60
call outString
push offset newline
call outString
push v_concat
call outString
push offset newline
call outString
push offset L_LIT_61
call outString
push offset newline
call outString
push v_length
call outUint
push offset newline
call outString
push offset L_LIT_62
call outString
push offset newline
call outString
push v_fromStr
call outUint
push offset newline
call outString
push offset L_LIT_63
call outString
push offset newline
call outString
push v_toStr
```

```
call outString
push offset newline
call outString

push 0
call ExitProcess
main ENDP
end main
```

Листинг 5 – Результат генерации ассемблерного кода