МИНИСТЕРСТВО ОБРАЗОВАНИЯ РЕСПУБЛИКИ БЕЛАРУСЬ

Учреждение образования «БЕЛОРУССКИЙ ГОСУДАРСТВЕННЫЙ ТЕХНОЛОГИЧЕСКИЙ УНИВЕРСИТЕТ»

Факультет Информационных Технологий
Кафедра Программной инженерии
Специальность 1-40 01 01 Программное обеспечение информационных технологий
Специализация Программирование интернет-приложений

ПОЯСНИТЕЛЬНАЯ ЗАПИСКА К КУРСОВОМУ ПРОЕКТУ НА ТЕМУ:

«Разработка компилятора KDV-2020»

Выполнил студент Каминский Дмитрий Викторович
(Ф.И.О.)

Руководитель проекта ст.пр. Наркевич Аделина Сергеевна
(учен. степень, звание, должность, подпись, Ф.И.О.)

Заведующий кафедрой к.т.н., доц. Пацей Наталья Владимировна
(учен. степень, звание, должность, подпись, Ф.И.О.)

Консультанты ст.пр. Наркевич Аделина Сергеевна
(учен. степень, звание, должность, подпись, Ф.И.О.)

Нормоконтролер ст.пр. Наркевич Аделина Сергеевна
(учен. степень, звание, должность, подпись, Ф.И.О.)

Курсовой проект защищен с оценкой

Минск 2020

Оглавление

Введение

Основной задачей курсового проекта является разработка компилятора для моего языка программирования – KDV-2020. Он предназначен для выполнения простейших операций и арифметических действий над числами.

Компилятор – программа, основной задачей которой является трансляции программы с одного языка программирования (в данном случае KDV-2020) в программу на языке ассемблера.

Процесс компиляции состоит из 2 частей: анализа и синтеза. Анализ – процесс, при котором исходная программа разбивается на составные части и накладывается на грамматическую структуру. Синтез – процесс, при котором строится требуемая целевая программа на основе промежуточного представления и информации из таблицы лексем. Мой исходный код транслируется в язык ассемблера. Мой компилятор состоит из следующих составных частей:

- лексический анализатор;
- синтаксический анализатор;
- семантический анализатор;
- генератор кода в язык ассемблера.

Исходя из цели курсового проекта, были определены следующие задачи:

- разработка спецификации языка программирования;
- разработка структуры транслятора;
- разработка лексического анализатора;
- разработка синтаксического анализатора;
- разработка семантического анализатора;
- преобразование выражений;
- генерация кода в язык ассемблера;
- тестирование компилятора.

Решения каждой из указанных задач представлены в соответствующих главах курсового проекта, а именно:

В первой главе определена спецификация языка программирования, т.е. синтаксис и семантика языка.

Во второй главе представлена структура транслятора, а именно, перечислены основные компоненты транслятора, их назначение и принципы взаимодействия, перечень протоколов, формируемых транслятором и содержимое протоколов.

В третьей главе показана работа лексического анализатора, порождающего таблицы лексем и идентификаторов, а также ошибки, которые обрабатывает этот анализатор.

В четвертой главе речь идет о синтаксическом анализаторе, задачей которого является синтаксический разбор текста с распечаткой протокола разбора,

дерева разбора на основе таблицы лексем и ошибки, обрабатываемые синтаксический анализатором.

В пятой главе описан семантический анализатор, его работа и обрабатываемые ошибки.

В шестой главе описаны преобразования выражений, допускаемых языком и приведена часть протокола для контрольного примера, которая отображает результаты преобразования выражений в обратную польскую запись.

В седьмой главе представлена генерация кода, где из промежуточного представления происходит генерация кода на целевом языке.

В восьмой главе описан процесс тестирования транслятора.

1. Спецификация языка программирования

1.1. Характеристика языка программирования

Язык программирования KDV-2020 является процедурным, строго типизированным, не объектно-ориентированным, компилируемым.

Процедурный язык программирования – язык в котором используется метод разбиения на отдельные связанные между собой модули – подпрограммы, также называемые процедуры и функции.

Строго типизированный язык программирования – язык, в котором переменные привязаны к конкретным типам данных. Такой язык не позволяет использовать в выражениях различные типы данных и не выполняет неявных преобразований.

Объектно-ориентированный язык программирования – язык, построенный на принципах объектно-ориентированного программирования(ооп). В основе ооп лежит понятие объекта – некоторой сущности, которая объединяет в себя и данные(поля), и выполняемые объектом действия(методы).

Компилируемый язык программирования – язык программирования, исходный код которого преобразуется компилятором в исходный код на целевом языке программирования.

1.2. Алфавит языка

Алфавит языка KDV-2020 состоит из следующих множеств символов:

- латинские символы верхнего и нижнего регистра: {A, B, C, … , Z, a, b, c… , z};
- цифры: {0, 1, 2, 3, 4, 5, 6, 7, 8, 9};
- специальные символы: {+, - , *, /, $, ‘, &, ?, ~, !, >, <, };
- знаки пунктуации языка{() {} , ; =};
- пробельные символы: пробел, символ табуляции, символ перехода на новую строку

1.3. Применяемые сепараторы

Применяемые в языке KDV-2020 сепараторы представлены в таблице 1.1.

Таблица 1.1 – применяемые сепараторы

Сепаратор	Назначение
;	Разделитель программных конструкций
,	Разделитель параметров функции
{}	Блок функций
()	Блок параметров функции, приоритет арифметических операция, блок условных операций
[]	Блок операторов цикла и условных операторов
+,-,/,*	Арифметические операции
=	Оператор присваивания
>,<,!,~,$,&	Логические операции
?	Символ комментария

1.4. Применяемые кодировки

Для написания программ на языке KDV-2020 использовалась кодировка Windows-1251, содержащая символы латинского и русского алфавитов, а также специальные символы.

Кодовая таблица Windows-1251

	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0				◌	Ё	§	Є	·		°						
		1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
1																
	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
2		!	"	#	$	%	&		(	)	*	+				/
	32	33	34	35	36	37	38	39	40	41	42	43	44	45	46	47
3	0	1	2	3	4	5	6	7	8	9	:	;	<	=	>	?
	48	49	50	51	52	53	54	55	56	57	58	59	60	61	62	63
4	@	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
	64	65	66	67	68	69	70	71	72	73	74	75	76	77	78	79
5	P	Q	R	S	T	U	V	W	X	Y	Z	[	\	]	^	_
	80	81	82	83	84	85	86	87	88	89	90	91	92	93	94	95
6	`	a	b	c	d	e	f	g	h	i	j	k	l	m	n	o
	96	97	98	99	100	101	102	103	104	105	106	107	108	109	110	111
7	p	q	r	s	t	u	v	w	x	y	z	{	|	}]	~	□
	112	113	114	115	116	117	118	119	120	121	122	123	124	125	126	127
8	Ђ	Ѓ	‚	ѓ	„	…	†	‡		‰	Љ	‹	Њ	Ќ	Ћ	Џ
	128	129	130	131	132	133	134	135	136	137	138	139	140	141	142	143
9	ђ	'	'	"	"		–	—		™	љ	›	њ	ќ	ћ	џ
	144	145	146	147	148	149	150	151	152	153	154	155	156	157	158	159
A		Ў	ў	J	¤	Ґ	¦	§	Ё	©	Є	«		-	®	Ї
	160	161	162	163	164	165	166	167	168	169	170	171	172	173	174	175
B	°	±	І	і	ґ	µ		·	ё	№	є	»	j	S	s	ї
	176	177	178	179	180	181	182	183	184	185	186	187	188	189	190	191
C	А	Б	В	Г	Д	Е	Ж	З	И	Й	К	Л	М	Н	О	П
	192	193	194	195	196	197	198	199	200	201	202	203	204	205	206	207
D	Р	С	Т	У	Ф	Х	Ц	Ч	Ш	Щ	Ъ	Ы	Ь	Э	Ю	Я
	208	209	210	211	212	213	214	215	216	217	218	219	220	221	222	• 223
E	а	б	в	г	д	е	ж	з	и	й	к	л	м	н	о	п
	224	225	226	227	228	229	230	231	232	233	234	235	236	237	238	239
F	р	с	т	у	ф	х	ц	ч	ш	щ	ъ	ы	ь	э	ю	я
	240	241	242	243	244	245	246	247	248	249	250	251	252	253	254	255

1.5. Типы данных

В языке программирования KDV-2020 есть 2 типа данных: без знаковый целочисленный (4 байта), символьный тип данных. Описание данных типов представлено в таблице 1.2

Тип данных	Описание
ohela	Фундаментальный тип данных. В памяти занимает 4 байта. Используется для работы с целочисленными значениями. Принимает значения в диапазоне от 0 до 4 294 967 295 Поддерживаемые *арифметические* операции: +(бинарный) – оператор сложения; -(бинарный) – оператор вычитания; *(бинарный) – оператор умножения; /(бинарный) – оператор деления; = - присваивание значения; *логические* операции < - “меньше” > - “больше” != - оператор проверки на неравенство ~ - оператор проверки на равенство $ - “больше либо равно” & - “меньше либо равно”
symb	Фундаментальный тип данных. Предусмотрен для работы с символами. В памяти занимает 1 байт. Максимальное количество символов – 1. Инициализируется по умолчанию символом нулевой длины ‘ ’.

1.6. Преобразование типов данных

В языке KDV – 2020 преобразование типов данных не поддерживается. Язык является строго типизированным.

1.7. Идентификаторы

Идентификаторы должны начинаться только с символов латинского алфавита. Максимальный размер идентификатора- 15. Идентификаторы, объявленные внутри функции получают префикс, максимальный размер которого составляет 15 дополнительных символов. В случае превышения данного размера, идентификатор усекается до 15 символов. Идентификаторы не должны совпадать с ключевыми словами.

1.8. Литералы

С помощью литералов происходит инициализация переменных. Все литералы в языке KDV-2020 являются rvalue. В языке KDV – 2020 есть 2 типа литералов: целые (десятичные и шестнадцатеричные), а также строковые. Описание литералов представлено в таблице 1.3.

Таблица 1.3. – Описание литералов

Тип литерала	Описание
Литералы целого типа	Числа, представленные в шестнадцатеричной либо десятичной системе исчисления
Символьный литерал	1 символ, заключенный в ‘ ’ .

1.9. Объявление данных

Для объявления переменной используется ключевое слово dekl, после которого указывается типа данных и имя идентификатора. Допускается инициализация при объявлении.

Пример объявления числового типа с инициализацией:

dekl ohela Ovariable = 1;

dekl symb Svariable = ‘s’;

Для объявления функции требуется ключевое слово fung, после которого указывается тип возвращаемого значения функции и имя функции. После этого обязателен список параметров и тело функции.

1.10. Инициализация данных

При объявлении переменной допускается ее инициализация, при этом ей будет присвоено значение строкового либо числового литерала, стоящего после знака присваивания. Если переменная не инициализирована, то ей по умолчанию присваивается значение; для целочисленных данных – значение 0, для символьных – строка длины 0(‘ ’).

1.11. Инструкции языка

Инструкции языка представлены в таблице 1.4

Таблица 1.4. –Инструкции языка

Инструкция	Форма записи
Объявление переменной	<dekl><тип данных> <идентификатор>;
Объявление переменной с явной инициализацией	<dekl><тип данных> <идентификатор> = <значение>, где значение- литерал, идентификатор либо вызов функции
Объявление функции	**fung** <тип данных> <идентификатор>(<тип данных> <идентификатор>, …) { /программный блок/ **return** <идентификатор> / <литерал>; }
Вызов функции	<идентификатор> (<идентификатор>/<литерал>, …);
Вывод данных	**skriva** <идентификатор>/<литерал>;
Возврат значения	**return** <идентификатор>/<литерал>

1.12. Операции языка

Язык программирования KDV-2020 предусматривает операции, представленные в таблице 1.5. Приоритетность операций можно устанавливать, используя круглые скобки.

Таблица 1.5 – операции языка

Тип оператора	Оператор
Арифметические	+ - сложение - вычитание * - умножение / - деление
Логические	> - больше < - меньше ! – проверка на неравенство $ - больше либо равно ~ - равно & - меньше либо равно

1.13. Выражения и их вычисления

В языке KDV-2020 существует ряд правил составления выражений и их вычислений:

- Круглые скобки используются для смены приоритета операций;
- Выражение должно записываться в строку без переносов;
- Не допускается запись несколько подряд идущих арифметических операций
- Допускается использование в выражении вызова функции, возвращающей целочисленное значение.

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

1.14. Программные конструкции языка

Программа на языке KDV-2020 оформляется в виде главной функции и функций пользователя. Программные конструкции представлены в таблице 1.6

Таблица 1.6 – Основные конструкции языка

Конструкция	Реализация

Главная функция(точка входа в программу)	huvud ()
	{
	/программный блок/
	return 0;
	}

Функция пользователя	fung <тип данных>
	<идентификатор>(<тип данных>
	<идентификатор>)
	{
	/программный блок/
	return <идентификатор>
	}

Цикл	medan <условие> […]
Условный оператор	om <условие> […] annan […]

1.15. Область видимости переменных

Область видимости переменных: сверху вниз (как и в C++). Переменные, объявленные в одной функции, недоступны в другой. Все объявления и операции с переменными происходят внутри программного блока. Каждая переменная, объявленная внутри блока функции получают соответствующий префикс.

Глобальных переменных нет, а значит, каждая переменная должна быть объявлена внутри блока какой-либо функции. Параметры видны только внутри функции, в которую они были переданы.

1.16. Семантические проверки

В языке программирования KDV-2020 выполняются следующие семантические проверки:

- Наличие функции huvud - точки входа в программу;
- Наличие только одной точки входа в программу;
- Наличие закрывающих },),]
- Определение идентификатор перед использованием;
- Повторное объявление идентификатора;
- Соответствие типов функции и возвращаемого ею значения;
- Использование операций для целочисленных операций;
- Превышение максимального (5) количества параметров функции;
- Соответствие количества параметров у проинициализированной и вызываемой функции;
- Деление на 0;
- Отсутствие return в huvud;
- Присваивание переменной одного значение переменной другого типа.

1.17. Распределение оперативной памяти на этапе выполнения

Транслированный код использует две области памяти. В сегмент констант заносятся все литералы. В сегмент данных заносятся переменные и параметры функций. Локальная область видимости в исходном коде определяется за счет правил именования идентификаторов и регулируется их префиксами. Благодаря этому переменные остаются “локальными” несмотря на глобальную область видимости всех данных в языке ассемблера.

1.18. Стандартная библиотека и её состав

В языке KDV-2020 предусмотрена стандартная библиотека. Функции, входящие в состав библиотеки представлены в таблице 1.7. Стандартная библиотека подключается автоматически на этапе генерации кода.

Таблица 1.7 – Функции стандартной библиотеки KDV-2020

Имя функции	Возвращаемое значение	Параметры функции	Описание
Kraft()	ohela	ohela elem1, ohela elem2	Вычисление степени

Продолжение таблицы 1.7.

Rot()	ohela	ohela elem1,	Извлечение квадратного корня
Slump()	ohela	ohela elem1	Генерация случайного числа

Таблица 1.8 Дополнительные функции стандартной библиотеки

Функция на языке C++	Описание
void numout(unsigned int value)	Функция для вывода в стандартный поток значения целочисленного идентификатора/литерала.
void strlineout(char* line)	Функция для вывода в стандартный поток значения строкового, символьного идентификатора/литерала и перехода на новую строку.
void system_pause()	Функция которая запускается по окончанию программы и требует нажать любой символ для завершения программы
void printError(char* error)	Функция для вывода ошибок красным цветом

1.19. Ввод и вывод данных

Вывод данных осуществляется при помощи skriva.
skriva <идентификатор>/<литерал>;
skriva <выражение>;
Ввод данных не предусмотрен.

1.20. Точка входа

Точкой входа является функция **huvug.** Каждая программа должна содержать эту функцию, с первой операции которой начнется последовательно выполнение команд.

1.21. Препроцессор

Препроцессор в языке KDV-2020 не предусмотрен.

1.22. Соглашения о вызовах

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

- все параметры передаются через стек;
- память высвобождает вызываемый код;
- занесение в стек параметров идет справа налево.

1.23. Объектный код

KDV-2020 транслируется в язык ассемблера, а далее – в объектный код.

1.24. Классификация сообщений транслятора

Генерируемые транслятором сообщения должны давать максимально полную информацию о допущенной пользователем ошибке при написании программы. Сообщения транслятора представлены в таблице 1.9, а также в приложении А.

Таблица 1.9 Классификация ошибок

Номера ошибок	Характеристика
0 – 112	Системные ошибки
113 – 299	Ошибки лексического анализа
400 – 599	Ошибки семантического анализа
600 – 699	Ошибки синтаксического анализа
300-399, 700-999	Зарезервированные коды ошибок

1.25. Контрольный пример

Контрольный пример демонстрирует главные особенности языка KDV-2020: его фундаментальные типы, основные структуры, функции, процедуры, использование функций статической библиотеки. Исходный код контрольного примера представлен в приложении А.

2. Структура транслятора

2.1. Компоненты транслятора, их назначение и принципы взаимодействия

Транслятор преобразует программу, написанную на языке KDV-2020 в программу на языке ассемблера. Компонентами транслятора являются лексический, синтаксический и семантический анализаторы, а также генератор кода в язык ассемблера. Их взаимодействие показано на рисунке 2.1.

Рисунок 2.1 Структура транслятора языка программирования KDV-2020

Лексический анализ – первая фаза компиляции. Назначение лексического анализатор – нахождение ошибок лексики языка и формирование таблицы лексем и таблицы идентификаторов. Подробности в 3 главе.

Синтаксический анализ – часть компилятора, выполняющая синтаксический анализ, который заключается в проверке исходного кода на соответствие правилам грамматики. Входной информацией для синтаксического анализа является таблица лексем и таблица идентификаторов. Выходной информацией является дерево разбора. Подробности в 4 главе.

Семантический анализ в свою очередь является проверкой исходной программы на семантическую согласованность с определением языка, т.е. проверяет правильность текста исходной программы с точки зрения семантики. Подробности в 5 главе.

Генератор кода – этап компиляции, выполняющий генерацию ассемблерного кода на основе полученных данных на предыдущих этапах трансляции. Генератор кода принимает на вход таблицы идентификаторов и лексем и транслирует кода с языка KDV-2020, прошедший все предыдущие этапы компиляции в код на языке ассемблера. Подробности в главе 7.

2.2. Перечень входных параметров транслятора

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

Таблица 2.1 Входные параметры транслятора языка KDV-2020

Входной параметр	Описание параметра	Значение по умолчанию
-in:<путь к in-файлу>	Файл с исходным кодом на языке KDV-2020, имеющий расширение .txt	Не предусмотрено
-log:<путь к log-файлу>	Файл журнала для вывода протоколов работы программы.	Значение по умолчанию: <имя in-файла>.log
-out:<путь к out-файлу>	Выходной файл – результат работы компилятора. Содержит исходный код на языке ассемблера.	Значение по умолчанию: <имя in-файла>.asm

2.3. Протоколы, формируемые транслятором

В ходе работы программы формируются протоколы работы лексического, синтаксического и семантического анализаторов, которые содержат в себе перечень протоколов работы. В таблице 2.2 приведены протоколы, формулируемые транслятором и их содержимое.

Таблице 2.2 Протоколы, формируемые транслятором языка KDV-2020

Формируемый протокол	Описание выходного протокола
Файл журнала, заданный параметром "-log:"	Файл с протоколом работы транслятора языка программирования KDV-2020. Содержит ошибки, полученные и обработанные при трансляции кода.
Выходной файл, заданный параметром "-out:"	Результат работы программы – файл, содержащий исходный код на языке ассемблера.
Выходной файл Table_Of_Lexem	Результат работы лексического анализатора, содержит таблицу лексем
Выходной файл Table_Of_Identificators	Результат работы лексического анализатора, содержит таблицу идентификаторов
Выходной файл Text_After_Lex	Результат работы лексического анализатора, содержит код, полученный в результате лексического анализа

3. Разработка лексического анализатора

3.1. Структура лексического анализатора

Первая стадия работы компилятора – лексический анализ, а программа, которая ее реализует – лексический анализатор (сканер). На вход лексического анализатора подается исходный код входного языка. Лексический анализатор выделяет в этой последовательности простейшие конструкции языка. Лексический анализатор производит предварительный разбор текста, преобразующий единый массив текстовых символов в массив токенов.

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

Функции лексического анализатора:

- удаление “пустых” символов и комментариев. Если “пустые” символы (пробелы, символы табуляции и перехода на новую строку) и комментарии будут удалены лексическим анализатором, синтаксическому и семантическому анализатор не придется их обрабатывать, затрачивая дополнительные ресурсы.
- распознавание идентификаторов и ключевых слов;
- распознавание литералов;
- распознавание разделителей и знаков операций.

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

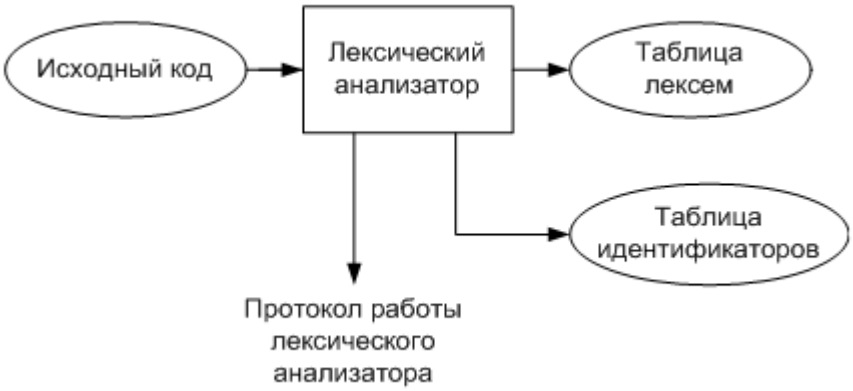


Рисунок 3.1. Структура лексического анализатора

3.2. Контроль входных символов

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

```
#define IN_CODE_TABLE {\
 /*0-15*/      IN::F, IN::F, IN::F, IN::F, IN::F, IN::F, IN::F, IN::F, IN::F, IN::TB, IN::N, IN::F, IN::F, IN::F, IN::F, IN::F,\
 /*16-31*/     IN::F, IN::F, IN::F, IN::F, IN::F, IN::F, IN::F, IN::F, IN::F, IN::F, IN::F, IN::F, IN::F, IN::F, IN::F, IN::F,\
 /*32-47*/     IN::S, IN::E, IN::F, IN::F, IN::E, IN::F, IN::E, IN::Q, IN::E, IN::E, IN::E, IN::E, IN::E, IN::E, IN::F, IN::E,\
 /*48-63*/     IN::T, IN::T, IN::T, IN::T, IN::T, IN::T, IN::T, IN::T, IN::T, IN::T, IN::F, IN::E, IN::E, IN::E, IN::E, IN::C,\
 /*64-79*/     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,\
 /*80-95*/     IN::T, IN::T, IN::T, IN::T, IN::T, IN::T, IN::T, IN::T, IN::T, IN::T, IN::T, IN::T, IN::T, IN::F, IN::T, IN::F, IN::F,\
 /*96-111*/    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,\
 /*112-127*/   IN::T, IN::T, IN::T, IN::T, IN::T, IN::T, IN::T, IN::T, IN::T, IN::T, IN::T, IN::T, IN::E,IN::F, IN::E, IN::E, IN::F,\
                                                                                                                              \
/*128-143*/    IN::F, IN::F, IN::F, IN::F, IN::F, IN::F, IN::F, IN::F, IN::F, IN::F, IN::F, IN::F, IN::F, IN::F, IN::F, IN::F, \
/*144-159*/    IN::F, IN::F, IN::F, IN::F, IN::F, IN::F, IN::F, IN::F, IN::F, IN::F, IN::F, IN::F, IN::F, IN::F, IN::F, IN::F, \
/*160-175*/    IN::F, IN::F, IN::F, IN::F, IN::F, IN::F, IN::F, IN::F, IN::F, IN::F, IN::F, IN::F, IN::F, IN::F, IN::F, IN::F, \
/*176-191*/    IN::F, IN::F, IN::F, IN::F, IN::F, IN::F, IN::F, IN::F, IN::F, IN::F, IN::F, IN::F, IN::F, IN::F, IN::F, IN::F, \
/*192-207*/    IN::F, IN::F, IN::F, IN::F, IN::F, IN::F, IN::F, IN::F, IN::F, IN::F, IN::F, IN::F, IN::F, IN::F, IN::F, IN::F, \
/*208-223*/    IN::F, IN::F, IN::F, IN::F, IN::F, IN::F, IN::F, IN::F, IN::F, IN::F, IN::F, IN::F, IN::F, IN::F, IN::F, IN::F, \
/*224-239*/    IN::F, IN::F, IN::F, IN::F, IN::F, IN::F, IN::F, IN::F, IN::F, IN::F, IN::F, IN::F, IN::F, IN::F, IN::F, IN::F, \
/*240-255*/    IN::F, IN::F, IN::F, IN::F, 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
Символьный литерал	Q
Символы выражения	E
Символ комментария	C
Символ пробела	S
Перевод строки	N
Пробел, табуляция	TB

3.3. Удаление избыточных символов

Избыточными считаются символы табуляции и пробелы.

Поэтому избыточные символы удаляются на этапе разбиения исходного кода на токены.

Описание алгоритма удаления избыточных символов:

1. Посимвольно считываем файл с исходным кодом программы;
2. При встрече пробела или знака табуляции игнорируем их для записи в таблицу лексем.

3.4. Перечень ключевых слов

Лексический анализатор преобразует исходный текст, заменяя лексические единицы на лексемы с целью создания промежуточного представления исходной программы. Соответствие токенов и лексем приведено в таблице 3.2.

Таблица 3.2 Соответствие токенов и сепараторов с лексемами

Токен	Лексема	Пояснение
ohela,symb	t	Названия типов данных языка.
Идентификатор	i	Длина идентификатора – 8 символов.
Литерал	l	Литерал любого доступного типа.
fung	f	Объявление функции.
Kraft	K	Вычисление степени
Rot	R	Вычисление квадратного корня
Slump	S	Генерация случайного числа
return	r	Выход из функции/процедуры.
huvug	h	Главная функция.
dekl	d	Объявление переменной.
skriva	s	Вывод данных.
om	o	Указывает начало условного оператора.
medan	c	Указывает начало цикла.
;	;	Разделение выражений.
,	,	Разделение параметров функций.
{	{	Начало блока.
}	}	Конец блока.
[	[	Начало блока цикла/условного оператора
]	]	Конец блока цикла/условного оператора
(	(	Передача параметров в функцию, приоритет операций.
)	)	Закрытие блока для передачи параметров, приоритет операций.
=	=	Знак присваивания.
+ - * /	v v v v	Знаки операций.
>< < & ! $ ~	b b b b b b	Знаки логических операторов

Пример реализации таблицы лексем представлен в приложении Б.

Каждому выражению соответствует детерминированный конечный автомат, по которому происходит разбор данного выражения. На каждый автомат в массиве подается токен и с помощью регулярного выражения оно записывается в таблицу лексем, если разбор был удачный. Если выражение является идентификатором или литералом, информация также заносится в таблицу лексем и в таблицу идентификаторов с дополнительной информацией. Структура конечного автомата и пример графа перехода конечного автомат изображены на рисунках 3.3 и 3.4 соответственно.

```
namespace FST
{
    struct RELATION     //ребро:символ -> вершина графа переходов КА
    {
        char symbol;    //символ перехода
        short nnode;    //номер смежной вершины
        RELATION(char c=0x00, short ns=NULL);
    };
    struct NODE //вершина графа переходов
    {
        short n_relation;//количество инцидентных ребер
        RELATION* relations;//инцидентные ребра
        NODE();
        NODE(short n, RELATION rel, ...);//количество ребер, список ребер
    };
    struct FST
    {
        char* string;   //цепочка
        short position;
        short nstates;  //количество состояний
        NODE* nodes;    //граф переходов
        short* rstates;//возможные состояния
        FST(char* s, short ns, NODE n, ...);
    };
    bool execute(FST& fst);
}
```

Рисунок 3.3 Структура конечного автомата

```
#define FST_HUVUD word,6,\
FST::NODE(1, FST::RELATION('h',1)),\
FST::NODE(1, FST::RELATION('u',2)),\
FST::NODE(1, FST::RELATION('v',3)),\
FST::NODE(1, FST::RELATION('u',4)),\
FST::NODE(1, FST::RELATION('d',5)),\
FST::NODE()
```

Рисунок 3.4 Пример графа перехода конечного автомата для токена huvud

3.5. Основные структуры данных

Основными структурами данных лексического анализатора являются таблица лексем и таблица идентификаторов. Таблица лексем содержит номер лексемы, лексему (lexema), полученную при разборе, номер строки в исходном коде (sn), и номер в таблице идентификаторов, если лексема является идентификаторов (idxTI). Таблица идентификаторов содержит имя идентификатора (id), номер в таблице лексем (idxfirstLE), тип данных (iddatatype), тип идентификатора (idtype) и значение (или параметры функций) (value). Код C++ со структурой таблицы лексем представлен на рисунке 3.3. Код C++ со структурой таблицы идентификаторов представлен на рисунке 3.4.

```
struct Entry
{
    char lexema;//лексема
    char arithmeticSymbol;
    int sn;                     //номер строки в исходном тексте
    int idxTI;                  //индекс в таблице идентификаторов или LT_TI_NULLIDX
    int priority;               //приоритет операции
};
struct LexTable//экземпляр таблицы лексем
{
    int maxsize;//емкость таблицы лексем < LT_MAXSIZE
    int size;//текущий размер таблицы лексем <maxsize
    Entry* table;//массив строк таблицы лексем
};
```

Рисунок 3.3 Структура таблицы лексем

```
struct Entry                        //строка таблицы идентификаторов
{
    int idxfirstLE;                             //индекс первой строки в таблице лексем
    char id[ID_MAXSIZE+1];                      //идентификатор(автоматически усекается до ID_MAXSIZE)
    char prefix[PREFIX_SIZE+1];         //используется для определения области видимости
    IDDATATYPE iddatatype;                      //тип данных
    IDTYPE idtype;                              //тип идентификатора
    union
    {
        int vint;
        struct
        {
            char ken;                       //количество символов в string
            char str[TI_STR_MAXSIZE - 1];   //символы string
        }vstr[TI_STR_MAXSIZE];              //значение string
    }value;//значение идентификатора
};
struct IdTable              //экземпляр таблицы идентификаторов
{
    int maxsize;            //емкость таблицы идентификаторов <TI_MAXSIZE
    int size;               //текущий размер таблицы идентификаторов < maxsize
    Entry* table;           //массив строк таблицы идентификаторов
};
```

Рисунок 3.4 Структура таблицы идентификаторов

3.6. Принцип обработки ошибок

Для обработки ошибок лексический анализатор использует таблицу с сообщениями. Структура сообщений содержит информацию о этапе компиляции, на котором произошла ошибка, номере сообщения, номере строки и позицию, где было вызвано сообщение в исходном коде и информацию об ошибке. При возникновении сообщения, лексический анализатор прекращает работу. Перечень сообщений представлен на рисунке 3.5.

```
ERROR_ENTRY(113,"[LEX] Error creating token table file/ Ошибка при создании файла таблицы лексем"),
ERROR_ENTRY(114,"[LEX] Error creating ID table file/ Ошибка при создании файла таблицы идентификаторов"),
ERROR_ENTRY(115,"Error creating file after processing/ Ошибка при создании файла после обработки"),                ERROR_ENTRY_NODEF(116),ERROR_ENTRY_NODEF(117),ERROR_ENTRY_NODEF

ERROR_ENTRY(120, "[LEX] Token size exceeded/ Превышен размер лексемы"),
ERROR_ENTRY(121, "[LEX] Trying to get outside the table/ Попытка попасть за пределы таблицы"),
ERROR_ENTRY(122, "[LEX] Missing closing quote/ Отсутствует закрывающая кавычка"),
ERROR_ENTRY(123, "[LEX] The maximum size of the identity table has been exceeded/ Превышен максимальный размер таблицы идентификаторов"),
ERROR_ENTRY(124, "[LEX] The maximum size of the lexem table has been exceeded/ Превышен максимальный размер таблицы лексем"),
ERROR_ENTRY(125, "[LEX] Invalid start character of identifier/ Недопустимый символ начала идентификатора"),
ERROR_ENTRY(126, "[LEX] Unable to recognize expression/ Невозможно распознать выражение"),
ERROR_ENTRY(127, "[LEX] Attempting to infer an undefined data type/ Попытка вывести неопределенный тип данных"),
ERROR_ENTRY(128, "[LEX] Attempting to infer an undefined identifier type/ Попытка вывести неопределенный тип идентификатора"),
ERROR_ENTRY(129, "[LEX] Overriding/ Переопределение"),
ERROR_ENTRY(130, "[LEX] Exceeded size of character literal [1]/Превышен размер символьного литерала[1]"),
ERROR_ENTRY(131, "[LEX] Error, no such function exists (possibly truncation to 15 characters)/Ошибка, такой функции не существует(возможно, произошло усечение до 15 символов)"),
ERROR_ENTRY(132, "[LEX] Error, you need to declare a variable before using it/Ошибка, необходимо объявить переменную перед ее использованием"),
```

Рисунок 3.5 – Сообщения лексического анализатора

3.7. Структура и перечень сообщений лексического анализатора

Ошибки, возникающие в процессе трансляции программы, фиксируются в протокол, заданный входными параметрами. В случае возникновения ошибок происходи их протоколирование с номер ошибки и диагностическим сообщением. Если в процесс анализ находится одна ошибка, анализ прекращается.

3.8. Параметры лексического анализатора

Результаты работы лексического анализатора, а именно таблица лексем и таблица идентификаторов выводятся в файлы с соответствующими названиями.

3.9. Алгоритм лексического анализа

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

Распознавание цепочек основывается на работе конечных автоматов.

3.10. Контрольный пример

Результат работы лексического анализатора в виде таблиц лексем и идентификаторов, соответствующих контрольному примеру, представлен в приложении Б.

4. Разработка синтаксического анализатора

4.1. Структура синтаксического анализатора

Синтаксический анализатор: часть компилятора, выполняющая синтаксический анализ, который заключается в проверке на соответствие правилам грамматики. На вход для синтаксического анализа подается таблица лексем и таблица идентификаторов. На основе этих таблиц будет построено дерево разбора, которое является результатом работы синтаксического анализатора.

Описание структуры синтаксического анализатора языка представлено на рисунке 4.1.

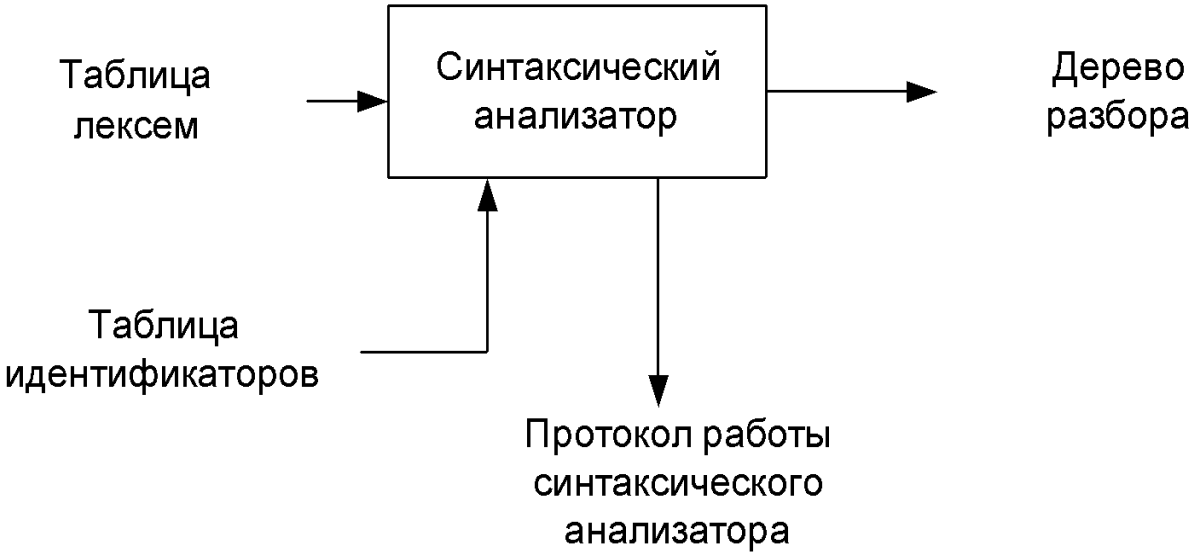


Рисунок 4.1 Структура синтаксического анализатора.

4.2. Контекстно-свободная грамматика, описывающая синтаксис языка

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

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

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

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

S – начальный символ грамматики, являющийся нетерминалом.

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

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

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

Описание нетерминальных символов содержится в таблице 4.1.

Таблица 4.1 Таблица правил переходов нетерминальных символов

Символ	Правила	Какие правила порождает
S	S->h{NrE;} S->fti(F){NrE;}S S->tfi(F){NrE;}	Стартовые правила, описывающее общую структуру программы
N	N->dti; N->dti;N N->dti=E; N->dti=E;N N->dti=M; N->dti=M;N N->i=E; N->i=E;N N->i=W; N->i=W;N N->oBC; N->cBK N->rR N->zP	Правила для построения операторов
E	E->i E->l E->(E) E->i(W) E->iM E->lM E->(E)M E->i(W)M E->k(i,l) E->k(i,i) E->k(l,l) E->k(l,i) E->u(i) E->u(l) E->q(i) E->q(l)	Правила для арифметических выражений
M	M->vE M->vEM	Правила построения математических выражений
F	F->ti,E F->ti	Правила для списка параметров функции
W	W->i W->l W->i,W	Правила для параметров вызываемых функций

Продолжение таблицы 4.1.

	W->l,W	
B	B->ibi B->ibl B->lbi B->ibi	Правила составления булевого выражения
C	C->[N] C->[N]N C->[N]a[N] C->[N]a[N]N	Правила составления условного оператора
K	C->[N] C->[N]N	Правила составления оператора цикла
P	P->l; P->i; P->l;N P->i;N	Правила составления оператора вывода
R	R->l; R->i; R->l;N R->i;N	Правила составления return

4.3. Построение конечного магазинного автомата

Конечный автомат с магазинной памятью представляет собой семерку $M = \langle Q, V, Z, \delta, q_0, z_0, F \rangle$. Подробное описание компонентов магазинного автомата представлено в таблице 4.2.

Таблица 4.2 – Описание компонентов магазинного автомата

Компонента	Определение	Описание
Q	Множество состояний автомата	Состояние автомата представляет из себя структуру, содержащую позицию на входной ленте, номера текущего правила и цепочки и стек автомата
V	Алфавит входных символов	Алфавит представляет из себя множества терминальных и нетерминальных символов, описание которых содержится в таблица 3.1 и 4.1.
Z	Алфавит специальных магазинных символов	Алфавит магазинных символов содержит стартовый символ и маркер дна стека (представляет из себя символ $)

Продолжение таблицы 4.2.

δ	Функция переходов автомата	Функция представляет из себя множество правил грамматики, описанных в таблице 4.1.
q_0	Начальное состояние автомата	Состояние, которое приобретает автомат в начале своей работы. Представляется в виде стартового правила грамматики
z_0	Начальное состояние магазина автомата	Символ маркера дна стека $
F	Множество конечных состояний	Конечные состояние заставляют автомат прекратить свою работу. Конечным состоянием является пустой магазин автомата и совпадение позиции на входной ленте автомата с размером ленты

4.4. Основные структуры данных

Основные структуры данных синтаксического анализатора представляются в виде структуры магазинного конечного автомата, выполняющего разбор исходной ленты, и структуры грамматики Грейбах, описывающей синтаксические правила языка KDV-2020. Правила языка KDV-2020 в приложении В.

4.5. Описание алгоритма синтаксического разбора

Принцип работы автомата следующий:
В магазин записывается стартовый символ;

2. На основе полученных ранее таблиц формируется входная лента;
3. Запускается автомат;
4. Выбирается цепочка, соответствующая нетерминальному символу, записывается в магазин;
5. Если терминалы в стеке и в ленте совпадают, то данный терминал удаляется из ленты и стека. Иначе возвращаемся в предыдущее сохраненное состояние и выбираем другую цепочку нетерминала;
6. Если в магазине встретился нетерминал, переходим к пункту 4;
7. Если наш символ достиг дна стека, и лента в этот момент пуста, то синтаксический анализ выполнен успешно. Иначе генерируется исключение.

4.6. Структура и перечень сообщений синтаксического анализатора

Перечень сообщений синтаксического анализатора представлен на рисунке 4.3.

```
ERROR_ENTRY(600, "[SYNT] Invalid program structure/ Неверная структура программы"),
ERROR_ENTRY(601, "[SYNT] Erroneous operator/ Ошибочный оператор"),
ERROR_ENTRY(602, "[SYNT] Expression error/ Ошибка в выражении"),
ERROR_ENTRY(603, "[SYNT] Error in function parameters/ Ошибочный арифметический оператор"),
ERROR_ENTRY(604, "[SYNT] Error in the parameters of the called function/ Ошибка в параметрах вызываемой функции"),
ERROR_ENTRY(605, "[SYNT] Error in function parameters when defining/ Ошибка в параметрах функции при определении"),
ERROR_ENTRY(606, "[SYNT] Error in using boolean expressions/ Ошибка в использовании булевых выражений"),
ERROR_ENTRY(607, "[SYNT] / Ошибка в условном операторе"),
ERROR_ENTRY(608, "[SYNT] / Ошибка в операторе цикла"),
ERROR_ENTRY(609, "[SYNT] / Ошибка в операторе вывода на экран"),
ERROR_ENTRY(610, "[SYNT] / Ошибка в операторе, возвращающем значение"),
```

Рисунок 4.3 – Сообщения синтаксического анализатора

4.7. Параметры синтаксического анализатора и режим его работы

Входной информацией для синтаксического анализатора является таблица лексем и идентификаторов. Кроме того, используется описание грамматики в форме Грейбах. Результат работы синтаксического анализа а именно, построенное дерево разбора и протокол работы автомата с магазинной памятью выводится в файл работы программы.

4.8. Принцип обработки ошибок

Синтаксический анализатор выполняет разбор исходной последовательности лексем до тех пор, пока не дойдёт до конца цепочки лексем или не найдёт ошибку. Тогда анализ останавливается и красным цветом выводится сообщение об ошибке (если она найдена). Если в процессе анализа находятся более трёх ошибок, то анализ останавливается.

4.9. Контрольный пример

Результат работы синтаксического анализатора (дерево разбора и протокол работы автомата с магазинной памятью) приведен в приложении В.

5. Разработка семантического анализатора

5.1. Структура семантического анализатора

Семантический анализатор принимает на вход результат работы лексического анализатора, то есть таблица лексем и таблица идентификаторов. Проверка на повторную инициализацию выполнена на этапе лексического анализа. Структура данного семантического анализатора представлена на рисунке 5.1.

Рисунок 5.1. Структура семантического анализатора

5.2. Функции семантического анализатора

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

5.3. Структура и перечень сообщений семантического анализатора

Сообщения, формируемые семантическим анализатором, представлены на рисунке 5.2.

```
ERROR_ENTRY(400, "[SEM] Entry point missing/Отсутствует точка входа"),
ERROR_ENTRY(401, "[SEM] The number of entry points exceeds 1/Количество точек входа превышает 1"),
ERROR_ENTRY(402, "[SEM] Closing \'}\' missing /Отсутствует закрывающая \'}\' скобка"),
ERROR_ENTRY(403, "[SEM] Missing closing '\)'\/Отсутствует закрывающая '\)'\ скобка"),
ERROR_ENTRY(404, "[SEM] Identifier not defined/Идентификатор не определен"),
ERROR_ENTRY(405, "[SEM] Range exceeded for the specified type/Превышен диапазон значений указанного типа"),
ERROR_ENTRY(406, "[SEM] Invalid return type/Неверный тип возвращаемого значения"),
ERROR_ENTRY(407, "[SEM] Using operations for different data types/Использование операций для разных типов данных"),
ERROR_ENTRY(408, "[SEM] Using invalid operations for strings/Использование недопустимых для строк операций"),
ERROR_ENTRY(409, "[SEM] Max number of function parameters exceeded/Превышено максимальное количество параметров функции"),
ERROR_ENTRY(410, "[SEM] Invalid number of function parameters/Неверное количество параметров функции"),
ERROR_ENTRY_NODEF(411),
ERROR_ENTRY(412, "[SEM] Error, division by 0/Ошибка, деление на 0"),
ERROR_ENTRY(413, "[SEM] Error, return is not allowed in huvud/Ошибка, в huvud недопустимо наличие return"),
ERROR_ENTRY(414, "[SEM] Error, assignment to a variable of a different type is not allowed/Ошибка, присвоение переменной другого типа недопустимо"),
ERROR_ENTRY_NODEF(415,"[SEM] Error missing closing \']\'/ Ошибка, отсутствует закрывающая \']\'"),
```

Рисунок 5.2 – Перечень сообщений семантического анализатора

5.4. Принцип обработки ошибок

Ошибки, возникающие в процессе трансляции программы, фиксируются в протокол, заданный входными параметрами (если не указаны, то заданными по умолчанию). В случае возникновения ошибки происходит остановка анализа, в протокол выводится сообщение об ошибке.

5.5. Контрольный пример

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

Исходный код	Текст сообщения

fung ohela id{
ohela x = 9;
skriva x;
}
	Ошибка N400: [SEM] Entry point missing/Отсутствует точка входа Строка: 4

huvud{
dekl ohela x = 9;
return 0;
	Ошибка N402: [SEM] Closing curly brace missing /Отсутствует закрывающая фигурная скобка Строка: 3

huvud{
dekl ohela x = 9;
return 0;
}
huvud{
dekl symb y = ‘q’;
return 0;
}
	Ошибка N401: [SEM] The number of entry points exceeds 1/Количество точек входа превышает 1 Строка: 5

6. Вычисление выражений

6.1. Выражения, допускаемые языком

В языке KDV-2020 допускаются вычисления выражений целочисленного типа данных с поддержкой вызова функций внутри выражений. Приоритет операций представлен на таблице 6.1.

Таблица 6.1. Приоритеты операций

Операция	Значение приоритета
( )	0
*	2
/	2
+	1
-	1

6.2. Польская запись и принцип ее построения

Все выражения в языке KDV-2020 преобразовываются к обратной польской записи.

Польская запись – альтернативный способ записи арифметических выражений, преимущество которого состоит в отсутствии скобок. Существует 2 типа польской записи: инфиксная(прямая) и постфиксная(обратная). Отличие их от стандартного типа записи заключается в том, что знаки операций пишутся не между, а, соответственно, до либо после переменных.

Алгоритм построения польской записи:

исходная строка: выражение;

результирующая строка: польская запись;

стек: пустой;

исходная строка просматривается слева направо;

операнды переносятся в результирующую строку;

операция записывается в стек, если стек пуст;

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

отрывающая скобка помещается в стек;

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

6.3. Программная реализация обработки выражений

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

6.4. Контрольный пример

Таблица 6.2. Преобразование выражений к ПОЛИЗ

Выражение	Обратная польская запись для выражения
i[2]=(((l[3]+l[4])-i[0])*l[5])/l[6];	i[2]=l[3]l[4]+i[0]-l[5]*l[6]/
i[23]=(i[23]+l[26])*l[26]	i[23]=i[23]l[26]+l[26]*
i[3]=(((l[4]+l[5])-i[0])*l[6])	i[3]=l[4]l[5]+i[0]-l[6]*

7. Генерация кода

7.1. Структура генератора кода

Генератор кода принимает на вход таблицы лексем и идентификаторов, полученные в результате лексического анализа. В соответствии с таблицей лексем строится выходной файл на языке ассемблера, который будет являться результатом работы компилятора языка KDV-2020. В случае возникновения ошибок генерация кода не будет осуществляться. Структура генератора кода языка KDV-2020 представлена на рисунке 7.1.

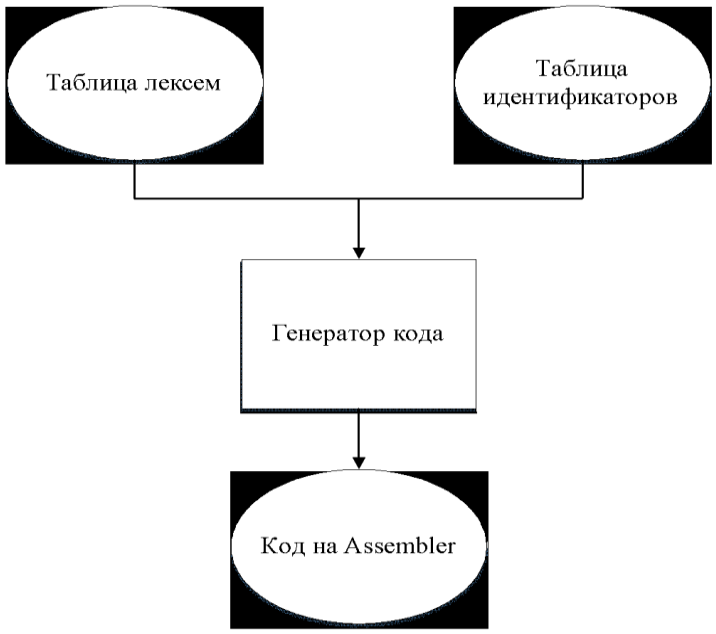


Рисунок 7.1 – Структура генератора кода

7.2. Представление типов данных в оперативной памяти

Элементы таблицы идентификаторов расположены в сегментах .data и .const языка ассемблера. Соответствия между типами данных идентификаторов языка KDV-2020 и языка ассемблера указаны в таблице 7.1.

Таблица 7.1 – Соответствия типов идентификаторов языка KDV-2020 и языка ассемблера

Тип идентификатора на языке KDV-2020	Тип идентификатора на языке ассемблера	Пояснение
ohela	dword	Хранит целочисленный тип данных.
symb	byte	Хранит символьный тип данных.

7.3. Статическая библиотека

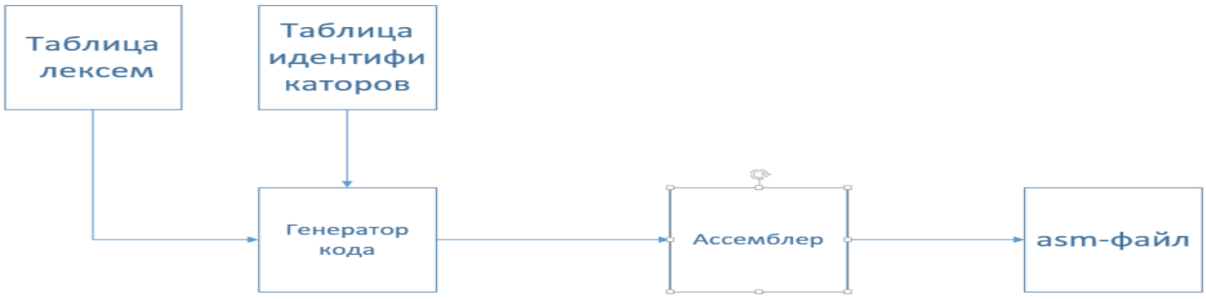
В языке KDV-2020 предусмотрена статическая библиотека. Статическая библиотека содержит функции, написанные на языке C++. Объявление функций статической библиотеки генерируется автоматически в коде ассемблера. Объявление функций статической библиотеки генерируется автоматически.

Таблица 7.3 – Функции статической библиотеки

Функция	Назначение
void strlineout(char* ptr)	Вывод на консоль строки ptr
void numout(unsigned int value)	Вывод на консоль целочисленной переменной value
void printError(char* ptr)	Вывод на консоль ошибки, обнаруженной в процессе выполнения кода, написанного на языке ассемблера
int Kraft(unsigned int a, unsigned int b)	Возведение числа a в степень b
void system_pause()	Ожидание нажатия клавиши пользователем
Slump(unsigned int a)	Генерация случайного числа в диапазоне от -a до a
Rot(unsigned int a)	Извлечение из числа a квадратного корня

7.4. Особенности алгоритма генерации кода

В языке KDV-2020 генерация кода строится на основе таблиц лексем и идентификаторов. Общая схема работы генератора кода представлена на рисунке 7.2



7.5. Входные параметры генератора кода

На вход генератору кода поступают таблицы лексем и идентификаторов исходного кода программы на языке KDV-2020. Результаты генератора кода выводятся в файл с расширением .asm.

7.6. Контрольный пример

Результат генерации ассемблерного кода на основе контрольного примера из приложения А приведен в приложении Д. Результат работы контрольного примера приведён на рисунке 7.2.

```
Kraft(12,3)
1728
Rot(15)
Slump(12)
Kraft(4,2)
16
V

1728
T

1760
```

Рисунок 7.2 Результат работы программы на языке KDV-2020

8. Тестирование транслятора

8.1. Общие положения

При возникновении ошибки на каком-либо этапе трансляции, она обрабатывается в главном файле программы: ошибка выводится на консоль красным цветом и записывается в файл логирования. Если ошибок обнаружено не было, текст консоли окрасится в зеленый цвет и будет выведена надпись об успешной компиляции файла

8.2. Результаты тестирования

Результаты тестирования приведены в таблице 8.1

Таблица 8.1 – Результаты тестирования транслятора

Исходный код	Диагностическое сообщение
мэйн	Ошибка 111: [SYST] Invalid character in source file (-in)/ Недопустимый символ в исходном файле (-in), строка 1, позиция 1

huvud{
dekl ohela i = a;
}
	Ошибка 404: [SEM] Identifier not defined/Идентификатор не определен, строка 1, позиция 0

ohela fung a(){
return 1;
}
	Ошибка 406: [SEM] Invalid return type/Неверный тип возвращаемого значения, строка 2, позиция 0

huvud{
ohela i = 3;
}
	Ошибка 601: строка 2, [SYNT] Erroneous operator/ Ошибочный оператор

huvud{dekl ohela i; dekl ohela i;}	Ошибка 129: [LEX] Overriding/ Переопределение, строка 4, позиция 12
ohela fung func(ohela a) {return 1;} main{ dekl ohela i = func(1,2);}	Ошибка 410: [SEM] Invalid number of function parameters/Неверное количество параметров функции Строка: 4

fung ohela func(ohela a)
{return 1;}
huvud{
dekl ohela i = func('s');
	Ошибка 406: [SEM] Invalid return type/Неверный тип возвращаемого значения, строка 2, позиция 0

Продолжение таблицы 8.1.

}	
huvud{dekl symb i = 's; }	Ошибка 122: [LEX] Missing closing quote/ Отсутствует закрывающая кавычка, строка 3, позиция 141, позиция 21
fung ohela func(ohela a) {return 4;} huvud{dekl ohela i = func('s');}	Ошибка 411: [SEM] Different types of function call parameters/Различные типы параметров вызова функции, строка 3, позиция 0
fung ohela func(ohela a) {return 4;} huvud{dekl ohela i = func('s);}	Ошибка 122: [LEX] Missing closing quote/ Отсутствует закрывающая кавычка, строка 3, позиция 26

Заключение

В ходе выполнения курсовой работы был разработан транслятор для языка программирования KDV-2020. Таким образом, были выполнены основные задачи данной курсовой работы:

- Сформулирована спецификация языка KDV-2020;
- Разработан лексический анализатор;
- Разработана контекстно-свободная, приведённая к нормальной форме Грейбах, грамматика;
- Разработан синтаксический анализатор;
- Разработан семантический анализатор;
- Разработан транслятор кода на язык ассемблера;
- Проведено тестирование транслятора.

Окончательная версия языка KDV-2020 включает:

- 2 типа данных;
- Поддержка операторов вывода;
- Возможность вызова функций стандартной библиотеки;
- Наличие 4 арифметических операторов для вычисления выражений;
- Наличие 6 булевых операторов для построения операторов цикла и условных операторов;
- Поддержка функций, процедур, операторов цикла и условных операторов;
- 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. Ахо, А. Компиляторы: принципы, технологии и инструменты / А. Ахо, Р. Сети, Дж. Ульман. – М.: Вильямс, 2003. – 768с.

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

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

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

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

Приложение А

Исходный код программы на языке KDV-2020

```
tung ohela function(ohela Vitya, ohela Mitya)
{
    dekl ohela power = Kraft(Bx100,2);
    skriva power;
    return power;
}
huvud
{
    dekl ohela Vitya = Ex14;
    dekl ohela Mitya = Hx10;
    ?Условный оператор?
    om Vitya < Mitya
    [
        Vitya = Kraft(Vitya,3);
        skriva Vitya;
    ]
    annan
    [
        Mitya = Kraft(Mitya,2);
        skriva Mitya;
    ]
    dekl ohela sqrt = Mitya + Rot(15);
    dekl ohela random = Slump(12);
    dekl ohela Together = Mitya + Vitya + function(Vitya,Mitya);
    medan Vitya < Mitya
    [
        Vitya = Vitya+1;
    ]
    skriva 'V';
    skriva Vitya;
    skriva 'T';
    skriva Together;
}
```

```
ERROR_ENTRY(0,"[SYST] Invalid error code/ Недопустимый код ошибки"),ERROR_ENTRY(1, "[SYST] System crash/ Системный сбой"),ERROR_ENTRY_NODEF(2), ERROR_ENTRY_NODEF(3), ERROR_ENTRY_
ERROR_ENTRY_NODEF(6), ERROR_ENTRY_NODEF(7), ERROR_ENTRY_NODEF(8), ERROR_ENTRY_NODEF(9),
ERROR_ENTRY_NODEF10(10), ERROR_ENTRY_NODEF10(20), ERROR_ENTRY_NODEF10(30), ERROR_ENTRY_NODEF10(40), ERROR_ENTRY_NODEF10(50), ERROR_ENTRY_NODEF10(60), ERROR_ENTRY_NODEF10(70), ERR
ERROR_ENTRY(100,"[SYST] The -in parameter must be given/Параметр -in должен быть задан"),
ERROR_ENTRY_NODEF(101), ERROR_ENTRY_NODEF(102), ERROR_ENTRY_NODEF(103),
ERROR_ENTRY(104,"[SYST] Length of input parameter exceeded/Превышена длина входного параметра"),
ERROR_ENTRY_NODEF(105), ERROR_ENTRY_NODEF(106), ERROR_ENTRY_NODEF(107), ERROR_ENTRY_NODEF(108), ERROR_ENTRY_NODEF(109),
ERROR_ENTRY(110,"[SYST] Error opening source file (-in)/Ошибка при открытии файла с исходным кодом (-in)"),
ERROR_ENTRY(111,"[SYST] Invalid character in source file (-in)/ Недопустимый символ в исходном файле (-in)"),
ERROR_ENTRY(112,"[SYST] Error creating log file (-log)/ Ошибка при создании файла протокола (-log)"),
ERROR_ENTRY(113,"[LEX] Error creating token table file/ Ошибка при создании файла таблицы лексем"),
ERROR_ENTRY(114,"[LEX] Error creating ID table file/ Ошибка при создании файла таблицы идентификаторов"),
ERROR_ENTRY(115,"Error creating file after processing/ Ошибка при создании файла после обработки"),                ERROR_ENTRY_NODEF(116),ERROR_ENTRY_NODEF(117),ERROR_ENTRY_NODE

ERROR_ENTRY(120, "[LEX] Token size exceeded/ Превышен размер лексемы"),
ERROR_ENTRY(121, "[LEX] Trying to get outside the table/ Попытка попасть за пределы таблицы"),
ERROR_ENTRY(122, "[LEX] Missing closing quote/ Отсутствует закрывающая кавычка"),
ERROR_ENTRY(123, "[LEX] The maximum size of the identity table has been exceeded/ Превышен максимальный размер таблицы идентификаторов"),
ERROR_ENTRY(124, "[LEX] The maximum size of the lexem table has been exceeded/ Превышен максимальный размер таблицы лексем"),
ERROR_ENTRY(125, "[LEX] Invalid start character of identifier/ Недопустимый символ начала идентификатора"),
ERROR_ENTRY(126, "[LEX] Unable to recognize expression/ Невозможно распознать выражение"),
ERROR_ENTRY(127, "[LEX] Attempting to infer an undefined data type/ Попытка вывести неопределенный тип данных"),
ERROR_ENTRY(128, "[LEX] Attempting to infer an undefined identifier type/ Попытка вывести неопределенный тип идентификатора"),
ERROR_ENTRY(129, "[LEX] Overriding/ Переопределение"),
ERROR_ENTRY(130, "[LEX] Exceeded size of character literal [1]/Превышен размер символьного литерала[1]"),
ERROR_ENTRY(131, "[LEX] Error, no such function exists (possibly truncation to 15 characters)/Ошибка, такой функции не существует(возможно, произошло усечение до 15 символов)"),
ERROR_ENTRY(132, "[LEX] Error, you need to declare a variable before using it/Ошибка, необходимо объявить переменную перед ее использованием"),
ERROR_ENTRY_NODEF(133),ERROR_ENTRY_NODEF(134),ERROR_ENTRY_NODEF(135),ERROR_ENTRY_NODEF(136),ERROR_ENTRY_NODEF(137),ERROR_ENTRY_NODEF(138),ERROR_ENTRY_NODEF(139),ERROR_ENTRY_NODEF
ERROR_ENTRY_NODEF100(200), ERROR_ENTRY_NODEF100(300),
ERROR_ENTRY(400, "[SEM] Entry point missing/Отсутствует точка входа"),
ERROR_ENTRY(401, "[SEM] The number of entry points exceeds 1/Количество точек входа превышает 1"),
ERROR_ENTRY(402, "[SEM] Closing \'}\' missing /Отсутствует закрывающая \'}\' скобка"),
ERROR_ENTRY(403, "[SEM] Missing closing '\)'\/Отсутствует закрывающая '\)'\ скобка"),
ERROR_ENTRY(404, "[SEM] Identifier not defined/Идентификатор не определен"),
ERROR_ENTRY(405, "[SEM] Range exceeded for the specified type/Превышен диапазон значений указанного типа"),
ERROR_ENTRY(406, "[SEM] Invalid return type/Неверный тип возвращаемого значения"),
ERROR_ENTRY(407, "[SEM] Using operations for different data types/Использование операций для разных типов данных"),
ERROR_ENTRY(408, "[SEM] Using invalid operations for strings/Использование недопустимых для строк операций"),
ERROR_ENTRY(409, "[SEM] Max number of function parameters exceeded/Превышено максимальное количество параметров функции"),
ERROR_ENTRY(410, "[SEM] Invalid number of function parameters/Неверное количество параметров функции"),
ERROR_ENTRY_NODEF(411),
ERROR_ENTRY(412, "[SEM] Error, division by 0/Ошибка, деление на 0"),
ERROR_ENTRY(413, "[SEM] Error, return is not allowed in huvud/Ошибка, в huvud недопустимо наличие return"),
ERROR_ENTRY(414, "[SEM] Error, assignment to a variable of a different type is not allowed/Ошибка, присвоение переменной другого типа недопустимо"),
ERROR_ENTRY_NODEF(415,"[SEM] Error missing closing \']\'/ Ошибка, отсутствует закрывающая \']\'"),
ERROR_ENTRY(600, "[SYNT] Invalid program structure/ Неверная структура программы"),
ERROR_ENTRY(601, "[SYNT] Erroneous operator/ Ошибочный оператор"),
ERROR_ENTRY(602, "[SYNT] Expression error/ Ошибка в выражении"),
ERROR_ENTRY(603, "[SYNT] Error in function parameters/ Ошибочный арифметический оператор"),
ERROR_ENTRY(604, "[SYNT] Error in the parameters of the called function/ Ошибка в параметрах вызываемой функции"),
ERROR_ENTRY(605, "[SYNT] Error in function parameters when defining/ Ошибка в параметрах функции при определении"),
ERROR_ENTRY(606, "[SYNT] Error in using boolean expressions/ Ошибка в использовании булевых выражений"),
ERROR_ENTRY(607, "[SYNT] / Ошибка в условном операторе"),
ERROR_ENTRY(608, "[SYNT] / Ошибка в операторе цикла"),
ERROR_ENTRY(609, "[SYNT] / Ошибка в операторе вывода на экран"),
ERROR_ENTRY(610, "[SYNT] / Ошибка в операторе, возвращающем значение"),
```

Приложение Б

Таблица идентификаторов контрольного примера

```
Размер таблицы идентификаторов: 21
Номер идентификатора -> Идентификатор -> Тип данных -> Тип идентификатора -> Строка в тексте -> Значение
    1                       function                 ohela            function         1
    2             function::Vitya                ohela           parameter         1
    3             function::Mitya                ohela           parameter         1
    4             function::power                ohela            variable         3                                0
    5                       Kraft                ohela     outsideFunction         3                                0
    6                       L1                 ohela             literal         3                              4
    7                       L2                 ohela             literal         3                              2
    8                huvud::Vitya                ohela            variable         9                                0
    9                       L3                 ohela             literal         9                             12
   10                huvud::Mitya                ohela            variable        10                                0
   11                       L4                 ohela             literal        10                             16
   12                       L5                 ohela             literal        14                              3
   13                huvud::sqrt                 ohela            variable        22                                 0
   14                       Rot                ohela     outsideFunction        22                                0
   15                       L6                 ohela             literal        22                             15
   16                huvud::random                ohela            variable        23                                  0
   17                       Slump                ohela     outsideFunction        23                                0
   18                huvud::Together                ohela            variable        24                                   0
   19                       L7                 ohela             literal        27                              1
   20                       L8                 symb              literal        29
   21                       L9                 symb              literal        31
```

Лексемы, соответствующие контрольному примеру

```
Обработанный текст
1 fti[0](ti[1],ti[2])
2 {
3 dti[3]=K(l[5],l[6]);
4 Pi[3];
5 ri[3];
6 }
7
8 h{
9 dti[7]=l[8];
10 dti[9]=l[10];
11
12 oi[7]b
13 i[9][
14 i[7]=K(i[7],l[11]);
15 Pi[7];
16 ]
17
18 a[
19 i[9]=K(i[9],l[6]);
20 Pi[9];
21 ]
22 dti[12]=i[9]vR(l[14]);
23 dti[15]=S(l[8]);
24 dti[17]=i[9]vi[7]vi[0](i[7],i[9]);
25 ci[7]b
26 i[9][
27 i[7]=i[7]vl[18];
28 ]
29 Pl[19];
30 Pi[7];
31 Pl[20];
32 Pi[17];
33 }
```

Приложение В

Грамматика языка

```
Greibach greibach(NS('S'), TS('$'), 11,
    Rule(NS('S'), GRB_ERROR_SERIES + 0,
        3,  //Неверная структура программы
        Rule::Chain(4, TS('h'), TS('{'), NS('N'), TS('}')),
        Rule::Chain(13, TS('f'), TS('t'), TS('i'), TS('('), NS('F'), TS(')'), TS('{'), NS('N'), TS('r'), NS('E'), TS(';'), TS('}'), NS('S')),
        Rule::Chain(12, TS('f'), TS('t'), TS('i'), TS('('), NS('F'), TS(')'), TS('{'), NS('N'), TS('r'), NS('E'), TS(';'), TS('}'))

    ),

    Rule(NS('N'), GRB_ERROR_SERIES + 1,
        14, //ошибочный оператор
        Rule::Chain(4, TS('d'), TS('t'), TS('i'), TS(';')),
        Rule::Chain(5, TS('d'), TS('t'), TS('i'), TS(';'), NS('N')),

        Rule::Chain(6, TS('d'), TS('t'), TS('i'), TS('='), NS('E'), TS(';')),
        Rule::Chain(7, TS('d'), TS('t'), TS('i'), TS('='), NS('E'), TS(';'), NS('N')),

        Rule::Chain(6, TS('d'), TS('t'), TS('i'), TS('='), NS('M'), TS(';')),
        Rule::Chain(7, TS('d'), TS('t'), TS('i'), TS('='), NS('M'), TS(';'), NS('N')),

        Rule::Chain(4, TS('i'), TS('='), NS('E'), TS(';')),
        Rule::Chain(5, TS('i'), TS('='), NS('E'), TS(';'), NS('N')),
        Rule::Chain(4, TS('i'), TS('='), NS('W'), TS(';')),
        Rule::Chain(5, TS('i'), TS('='), NS('W'), TS(';'), NS('N')),

        Rule::Chain(3,  TS('o'), NS('B'), NS('C')),//

        Rule::Chain(3, TS('c'), NS('B'), NS('K')),
        Rule::Chain(3, TS('r'), NS('R')),
        Rule::Chain(2, TS('z'), NS('P'))
    ),
```

```
Rule(NS('E'), GRB_ERROR_SERIES + 2,
    16, //выражение
    Rule::Chain(1, TS('i')),
    Rule::Chain(1, TS('l')),
    Rule::Chain(3, TS('('), NS('E'), TS(')')),
    Rule::Chain(4, TS('i'), TS('('), NS('W'), TS(')')),
    Rule::Chain(2, TS('i'), NS('M')),
    Rule::Chain(2, TS('l'), NS('M')),
    Rule::Chain(4, TS('('), NS('E'), TS(')'), NS('M')),
    Rule::Chain(5, TS('i'), TS('('), NS('W'), TS(')'), NS('M')),
    Rule::Chain(6, TS('k'), TS('('), TS('i'),TS(','), TS('l'), TS(')')),
    Rule::Chain(6, TS('k'), TS('('), TS('l'),TS(','), TS('i'), TS(')')),
    Rule::Chain(6, TS('k'), TS('('), TS('i'),TS(','), TS('i'), TS(')')),
    Rule::Chain(6, TS('k'), TS('('), TS('l'),TS(','), TS('l'), TS(')')),
    Rule::Chain(4, TS('u'), TS('('), TS('i'), TS(')')),
    Rule::Chain(4, TS('u'), TS('('), TS('l'), TS(')')),
    Rule::Chain(4, TS('q'), TS('('), TS('i'), TS(')')),
    Rule::Chain(4, TS('q'), TS('('), TS('l'), TS(')'))
),
```

```
Rule(NS('M'), GRB_ERROR_SERIES + 3,
    2,  //Ошибочный оператор
    Rule::Chain(2, TS('v'), NS('E')),
    Rule::Chain(3, TS('v'), NS('E'), NS('M'))
),
Rule(NS('W'), GRB_ERROR_SERIES + 4, 4, //параметры вызываемой ф-ции
    Rule::Chain(1, TS('i')),
    Rule::Chain(1, TS('l')),
    Rule::Chain(3, TS('i'), TS(','), NS('W')),
    Rule::Chain(3, TS('l'), TS(','), NS('W'))
),
Rule(NS('F'), GRB_ERROR_SERIES + 5, 2,  // параметры функции при определении
    Rule::Chain(2, TS('t'), TS('i')),
    Rule::Chain(4, TS('t'), TS('i'), TS(','), NS('F'))
),
Rule(NS('B'), GRB_ERROR_SERIES + 6,
    4, //булевые выражения
    Rule::Chain(3, TS('i'), TS('b'), TS('i')),
    Rule::Chain(3, TS('i'), TS('b'), TS('l')),
    Rule::Chain(3, TS('l'), TS('b'), TS('i')),
    Rule::Chain(3, TS('l'), TS('b'), TS('l'))
),
Rule(NS('C'), GRB_ERROR_SERIES + 7,
    4, //Условный оператор

    Rule::Chain(3, TS('['), NS('N'), TS(']')),//
    Rule::Chain(4, TS('['), NS('N'), TS(']'), NS('N')),//
    Rule::Chain(7, TS('['), NS('N'), TS(']'), TS('a'), TS('['), NS('N'), TS(']')),//
    Rule::Chain(8, TS('['), NS('N'), TS(']'), TS('a'), TS('['), NS('N'), TS(']'), NS('N'))//
),
Rule(NS ('K'), GRB_ERROR_SERIES + 8,        //Цикл
    2,
    Rule::Chain(3, TS('['), NS('N'), TS(']')),
    Rule::Chain(4, TS('['), NS('N'), TS(']'), NS('N'))
),
Rule(NS('P'), GRB_ERROR_SERIES + 9,     //Выражения для вывода
    4,
    Rule::Chain(2, TS('i'),TS(';')),
    Rule::Chain(2, TS('l'), TS(';')),
    Rule::Chain(3, TS('l'), TS(';'),NS('N')),
    Rule::Chain(3, TS('i'), TS(';'), NS('N'))
),
Rule(NS('R'), GRB_ERROR_SERIES + 10,        //Выражения для return
    4,
    Rule::Chain(2, TS('i'),TS(';')),
    Rule::Chain(2, TS('l'), TS(';')),
    Rule::Chain(3, TS('l'), TS(';'),NS('N')),
    Rule::Chain(3, TS('i'), TS(';'), NS('N'))
)
```

Разбор исходного кода синтаксическим анализатором

```
 Шаг: Правило              Входная лента                Стек
0   : S->fti(F){NrE;}S     fti(ti,ti){dti=k(l,l);zi;    S$
0   : SAVESTATE:           1
0   :                      fti(ti,ti){dti=k(l,l);zi;    fti(F){NrE;}S$
1   :                      ti(ti,ti){dti=k(l,l);zi;r    ti(F){NrE;}S$
2   :                      i(ti,ti){dti=k(l,l);zi;ri    i(F){NrE;}S$
3   :                      (ti,ti){dti=k(l,l);zi;ri;    (F){NrE;}S$
4   :                      ti,ti){dti=k(l,l);zi;ri;}    F){NrE;}S$
5   : F->ti                ti,ti){dti=k(l,l);zi;ri;}    F){NrE;}S$
5   : SAVESTATE:           2
5   :                      ti,ti){dti=k(l,l);zi;ri;}    ti){NrE;}S$
6   :                      i,ti){dti=k(l,l);zi;ri;}h    i){NrE;}S$
7   :                      ,ti){dti=k(l,l);zi;ri;}h{    ){NrE;}S$
8   : TS_NOK/TNS_NS_NORULECHAIN/NS_NORULE
8   : RESTATE
8   :                      ti,ti){dti=k(l,l);zi;ri;}    F){NrE;}S$
9   : F->ti,F              ti,ti){dti=k(l,l);zi;ri;}    F){NrE;}S$
9   : SAVESTATE:           2
9   :                      ti,ti){dti=k(l,l);zi;ri;}    ti,F){NrE;}S$
10  :                      i,ti){dti=k(l,l);zi;ri;}h    i,F){NrE;}S$
11  :                      ,ti){dti=k(l,l);zi;ri;}h{    ,F){NrE;}S$
12  :                      ti){dti=k(l,l);zi;ri;}h{d    F){NrE;}S$
13  : F->ti                ti){dti=k(l,l);zi;ri;}h{d    F){NrE;}S$
13  : SAVESTATE:           3
13  :                      ti){dti=k(l,l);zi;ri;}h{d    ti){NrE;}S$
14  :                      i){dti=k(l,l);zi;ri;}h{dt    i){NrE;}S$
15  :                      ){dti=k(l,l);zi;ri;}h{dti    ){NrE;}S$
16  :                      {dti=k(l,l);zi;ri;}h{dti=    {NrE;}S$
17  :                      dti=k(l,l);zi;ri;}h{dti=l    NrE;}S$
18  : N->dti;              dti=k(l,l);zi;ri;}h{dti=l    NrE;}S$
18  : SAVESTATE:           4
1443: TS_NOK/TNS_NS_NORULECHAIN/NS_NORULE
1443: RESTATE
1443:                      l;zi;}                       P}$
1444: P->l;N               l;zi;}                       P}$
1444: SAVESTATE:           51
1444:                      l;zi;}                       l;N}$
1445:                      ;zi;}                        ;N}$
1446:                      zi;}                         N}$
1447: N->zP                zi;}                         N}$
1447: SAVESTATE:           52
1447:                      zi;}                         zP}$
1448:                      i;}                          P}$
1449: P->i;                i;}                          P}$
1449: SAVESTATE:           53
1449:                      i;}                          i;}$
1450:                      ;}                           ;}$
1451:                      }                            }$
1452:                                                   $
1453: LENTA_END
1454: ------>LENTA_END
--------------------------------------------------------------------------------
```

Дерево разбора

```
0   всего строк 136, синтаксический анализ выполнен без ошибок
0   : S->fti(F){NrE;}S
4   : F->ti,F
7   : F->ti
11  : N->dti=E;N
15  : E->k(l,l)
22  : N->zP
23  : P->i;
26  : E->i
29  : S->h{N}
31  : N->dti=E;N
35  : E->l
37  : N->dti=E;N
41  : E->l
43  : N->oBC
44  : B->ibi
47  : C->[N]a[N]N
48  : N->i=E;N
50  : E->k(i,l)
57  : N->zP
58  : P->i;
63  : N->i=E;N
65  : E->k(i,l)
72  : N->zP
73  : P->i;
76  : N->dti=E;N
80  : E->iM
81  : M->vE
82  : E->q(l)
87  : N->dti=E;N
91  : E->u(l)
96  : N->dti=E;N
100 : E->iM
101 : M->vE
102 : E->iM
103 : M->vE
104 : E->i(W)
106 : W->i,W
108 : W->i
111 : N->cBK
112 : B->ibi
115 : K->[N]N
116 : N->i=E;
118 : E->iM
119 : M->vE
120 : E->l
123 : N->zP
124 : P->l;N
126 : N->zP
127 : P->i;N
129 : N->zP
130 : P->l;N
132 : N->zP
133 : P->i;
```

Приложение Г

Программная реализация механизма преобразования в ПОЛИЗ

```
void workWithPolishNotation(LT::LexTable& lextable, IT::IdTable& idtable)
{
    for (int lextable_pos = 0; lextable_pos < lextable.size; lextable_pos++)
    {
        if (lextable.table[lextable_pos].lexema == LEX_EQUALS)
        {
            lextable_pos = PolishNotation(lextable_pos+1, lextable, idtable);
        }
    }
}
int PolishNotation(int lexPos,LT::LexTable& lextable,IT::IdTable idtable)
{
    std::queue<LT::Entry> queue;
    std::stack <LT::Entry> steck;
    int countParams = 0;
    bool isParam = false;
    int Lexindex = lexPos;
    while (lextable.table[Lexindex].lexema!=LEX_SEMICOLON)
    {
        switch (lextable.table[Lexindex].lexema)
        {
            case LEX_OPER:
            {
                if (steck.empty() || steck.top().arithmeticSymbol == LEX_LEFTHESIS)
                    steck.push(lextable.table[Lexindex]);
                else
                {
                    while (!steck.empty() && steck.top().priority >= lextable.table[Lexindex].priority)
                    {
                        queue.push(steck.top());
                        steck.pop();
                    }
                    steck.push(lextable.table[Lexindex]);
                }
                break;
            }
            case LEX_SLUMP:
```

```
    case LEX_KRAFT:
    case LEX_ROT:
    case LEX_LITERAL:
    case LEX_ID:
        if (IT::GetEntry(idtable,lextable.table[Lexindex].idxTI).idtype == IT::F
            || IT::GetEntry(idtable, lextable.table[Lexindex].idxTI).idtype == IT::K)
        {
            queue.push(lextable.table[Lexindex]);
        }
        else
            queue.push(lextable.table[Lexindex]);
        break;
    case LEX_LEFTHESIS:
    {
        steck.push(lextable.table[Lexindex]);
        break;
    }
    case LEX_REIGHTHESIS:
    {
        while (steck.top().arithmeticSymbol != LEX_LEFTHESIS)
        {
            queue.push(steck.top()); steck.pop();
        }
        steck.pop();
        break;
    case LEX_COMMA:
        bool isLeftBrace = true;
        if (steck.top().arithmeticSymbol == LEX_LEFTHESIS)
            isLeftBrace = false;
        else
        {
            queue.push(steck.top()); steck.pop();
        }
        break;
    }
}
Lexindex++;
```

```
    if (!steck.empty())
    {
        queue.push(steck.top()); steck.pop();
    }
    return ChangeLex(lexPos,Lexindex, lextable, queue);
}
int ChangeLex(int lexPos,int lexIndex, LT::LexTable&lextable, std::queue<LT::Entry> queue)
{
    int index = 0;
    LT::Entry entry;
    entry.sn = -1;
    entry.idxTI = -1;
    entry.lexema = 'N';
    for (int i = queue.size(); i < lexIndex - lexPos; i++)
        queue.push(entry);
    for (; !queue.empty(); index++)
    {
        lextable.table[lexPos + index] = queue.front();
        queue.pop();
    }
    return lexPos + index;
}
```

Приложение Д

Результат генерации кода контрольного примера в Ассемблер

```
.586
.model flat, stdcall
includelib libucrt.lib
includelib kernel32.lib
includelib ..\KDV-2020\Debug\Stand_Lib.lib
ExitProcess PROTO:DWORD

.stack 4096


EXTRN e:   proc
EXTRN Kraft: proc
EXTRN Rot: proc
EXTRN Slump: proc
EXTRN sinus:  proc
EXTRN cosinus: proc
EXTRN strlineout: proc
EXTRN system_pause: proc
EXTRN numout:proc
EXTRN printError:proc

.const
null_division BYTE 'ERROR:DIVISION BY ZERO', 0
overflow BYTE 'ERROR: VARIABLE OVERFLOW', 0
_L1 DWORD   4
_L2 DWORD   2
_L3 DWORD   12
_L4 DWORD   16
_L5 DWORD   3
_L6 DWORD   15
_L7 DWORD   1
_L8 BYTE    'V',0
_L9 BYTE    'T',0

.data
function_power  DWORD ?
huvud_Vitya DWORD ?
huvud_Mitya DWORD ?
```

```
huvud_sqrt  DWORD ?
huvud_random    DWORD ?
huvud_Together  DWORD ?

.code
function PROC function_Vitya : DWORD, function_Mitya : DWORD,

push _L2
push _L1
call Kraft
push eax

pop function_power

push function_power
call numout

push function_power
pop eax
ret

SOMETHINGWRONG:
push offset null_division
call printError
call system_pause
push -1
call ExitProcess
EXIT_OVERFLOW:
push offset overflow
call printError
call system_pause
push -2
call ExitProcess
function ENDP

main PROC

push _L3
```

```
pop huvud_Vitya

push _L4

pop huvud_Mitya
push huvud_Vitya
push huvud_Mitya
pop ebx
pop eax
cmp eax,ebx
jl beginIf0
jmp endIf0
beginIf0:
push _L5
push huvud_Vitya
call Kraft
push eax

pop huvud_Vitya

push huvud_Vitya
call numout
jmp endElse0
endIf0:
push _L2
push huvud_Mitya
call Kraft
push eax

pop huvud_Mitya

push huvud_Mitya
call numout
endElse0:

push huvud_Mitya
push _L6
call Rot
```

```
push eax
pop ebx
pop eax
add eax, ebx
jo EXIT_OVERFLOW
push eax

pop huvud_sqrt
push _L3
call Slump
push eax

pop huvud_random

push huvud_Mitya

push huvud_Vitya
pop ebx
pop eax
add eax, ebx
jo EXIT_OVERFLOW
push eax
push huvud_Mitya
push huvud_Vitya
call function
push eax
pop ebx
pop eax
add eax, ebx
jo EXIT_OVERFLOW
push eax

pop huvud_Together

cycle1:
push huvud_Vitya
push huvud_Mitya
pop ebx
```

```
pop eax
cmp eax,ebx
jg endcycle1

push huvud_Vitya

push _L7
pop ebx
pop eax
add eax, ebx
jo EXIT_OVERFLOW
push eax

pop huvud_Vitya

jmp cycle1
endcycle1:

push offset _L8
call strlineout

push huvud_Vitya
call numout

push offset _L9
call strlineout

push huvud_Together
call numout

push 0
call ExitProcess

SOMETHINGWRONG:
push offset null_division
call printError
call system_pause
push -1
```

```
call ExitProcess
EXIT_OVERFLOW:
push offset overflow
call printError
call system_pause
push -2
call ExitProcess
main ENDP
end main
```