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

ПОЯСНИТЕЛЬНАЯ ЗАПИСКА
К КУРСОВОМУ ПРОЕКТУ НА ТЕМУ:

«Разработка компилятора PAA-2022»

Выполнил студент Пузиков Алексей Алексеевич
(Ф.И.О. студента)

Руководитель проекта асс. Мущук Артур Николаевич
(учен. степень, звание, должность, подпись, Ф.И.О.)

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

Консультанты асс. Мущук Артур Николаевич
(учен. степень, звание, должность, подпись, Ф.И.О.)

Курсовой проект защищен с оценкой

Минск 2022

Содержание

Введение

В данном курсовом проекте мы разрабатывали компилятор для собственного языка программирования.

Текст программы должен быть оттранслирован в соответствующую последовательность команд, прежде чем он может быть выполнен компьютером. Эта трансляция сама может быть описана программой. Транслирующая программа называется компилятором, а текст, который должен транслироваться, называется исходным текстом.

Транслятор РАА-2022 состоит из следующих частей:
– лексический анализатор;
– синтаксический анализатор;
– семантический анализатор;
– генератор исходного кода на языке ассемблера

В соответствии с курсовым проектом были определены следующие задачи:
– разработка спецификации языка программирования;
– разработка структуры транслятора;
– разработка программной реализации лексического анализатора;
– разработка программной реализации синтаксического анализатора;
– разработка программной реализации семантического анализатора;
– разработка программной реализации преобразования выражений;
– разработка программной реализации генератора кода;
– выполнить тестирование, разработанного программного обеспечения.

1 Спецификация языка программирования

1.1 Характеристика языка программирования

Язык PAA-2022 – это универсальный, строго типизированный, процедурный, компилируемый язык. Он не является объектно-ориентированным. В языке отсутствует преобразование типов. В языке поддерживается 2 типа данных: целочисленный беззнаковый (uint) и строковый (string). В стандартной библиотеке имеются функции для работы со строковым типом данных: strCmp – возвращает результат лексикографического сравнения строк, strLen – возвращает длину строки.

1.2 Определение алфавита языка программирования

Алфавит языка PAA-2022 основан на кодировке Windows-1251.

Символы, используемые на этапе выполнения: [a…z], [A…Z], [0…9], [а…я], [А…Я], символы пробела, табуляции и перевода строки, спецсимволы: (){} , ; : + - / * % > < ! ' .

1.3 Применяемые сепараторы

Символы, которые являются сепараторами представлены в таблице 1.1.

Таблица 1.1 – Сепараторы

Разделители	Назначение
‘пробел’, ‘табуляция’, ‘переход на новую строку’	Разделяют входные лексемы.
+, -, *, /	Арифметические операторы. Используются в арифметических операциях.
=	Оператор присваивания. Используется для присваивания значения переменной.
<, >, <=, >=, ==, !=	Условные операторы. Используются для сравнения переменных и литералов.
( )	Блок параметров функции, так же указывает приоритет в арифметических операциях.
,	Разделяет параметры функции.
{ }	Блок функции или программной конструкции условного оператора.
;	Признак конца инструкции языка.
' '	Ограничивают строковый литерал.

1.4 Применяемые кодировки

Для написания исходного кода на языке программирования PAA-2022 используется кодировка Windows-1251, представленная на рисунке 1.1.

	00	01	02	03	04	05	06	07	08	09	0A	0B	0C	0D	0E	0F
00	NUL 0000	STX 0001	SOT 0002	ETX 0003	EOT 0004	ENQ 0005	ACK 0006	BEL 0007	BS 0008	HT 0009	LF 000A	VT 000B	FF 000C	CR 000D	SO 000E	SI 000F
10	DLE 0010	DC1 0011	DC2 0012	DC3 0013	DC4 0014	NAK 0015	SYN 0016	ETB 0017	CAN 0018	EM 0019	SUB 001A	ESC 001B	FS 001C	GS 001D	RS 001E	US 001F
20	SP 0020	! 0021	" 0022	# 0023	$ 0024	% 0025	& 0026	' 0027	(0028	) 0029	* 002A	+ 002B	, 002C	- 002D	. 002E	/ 002F
30	0 0030	1 0031	2 0032	3 0033	4 0034	5 0035	6 0036	7 0037	8 0038	9 0039	: 003A	; 003B	< 003C	= 003D	> 003E	? 003F
40	@ 0040	A 0041	B 0042	C 0043	D 0044	E 0045	F 0046	G 0047	H 0048	I 0049	J 004A	K 004B	L 004C	M 004D	N 004E	O 004F
50	P 0050	Q 0051	R 0052	S 0053	T 0054	U 0055	V 0056	W 0057	X 0058	Y 0059	Z 005A	[005B	\ 005C	] 005D	^ 005E	_ 005F
60	` 0060	a 0061	b 0062	c 0063	d 0064	e 0065	f 0066	g 0067	h 0068	i 0069	j 006A	k 006B	l 006C	m 006D	n 006E	o 006F
70	p 0070	q 0071	r 0072	s 0073	t 0074	u 0075	v 0076	w 0077	x 0078	y 0079	z 007A	{ 007B	| 007C	} 007D	~ 007E	DEL 007F
80	Ђ 0402	Ѓ 0403	‚ 201A	ѓ 0453	„ 201E	… 2026	† 2020	‡ 2021	€ 20AC	‰ 2030	Љ 0409	‹ 2039	Њ 040A	Ќ 040C	Ћ 040B	Џ 040F
90	ђ 0452	‘ 2018	’ 2019	“ 201C	” 201D	• 2022	– 2013	— 2014		™ 2122	љ 0459	› 203A	њ 045A	ќ 045C	ћ 045B	џ 045F
A0	NBSP 00A0	Ў 040E	ў 045E	Ј 0408	¤ 00A4	Ґ 0490	¦ 00A6	§ 00A7	Ё 0401	© 00A9	Є 0404	« 00AB	¬ 00AC	- 00AD	® 00AE	Ї 0407
B0	° 00B0	± 00B1	І 0406	і 0456	ґ 0491	µ 00B5	¶ 00B6	· 00B7	ё 0451	№ 2116	є 0454	» 00BB	ј 0458	Ѕ 0405	ѕ 0455	ї 0457
C0	А 0410	Б 0411	В 0412	Г 0413	Д 0414	Е 0415	Ж 0416	З 0417	И 0418	Й 0419	К 041A	Л 041B	М 041C	Н 041D	О 041E	П 041F
D0	Р 0420	С 0421	Т 0422	У 0423	Ф 0424	Х 0425	Ц 0426	Ч 0427	Ш 0428	Щ 0429	Ъ 042A	Ы 042B	Ь 042C	Э 042D	Ю 042E	Я 042F
E0	а 0430	б 0431	в 0432	г 0433	д 0434	е 0435	ж 0436	з 0437	и 0438	й 0439	к 043A	л 043B	м 043C	н 043D	о 043E	п 043F
F0	р 0440	с 0441	т 0442	у 0443	ф 0444	х 0445	ц 0446	ч 0447	ш 0448	щ 0449	ъ 044A	ы 044B	ь 044C	э 044D	ю 044E	я 044F

Рисунок 1.1 – Алфавит входных символов

1.5 Типы данных

В языке РАА-2022 есть 2 типа данных: целочисленный беззнаковый и строковый. Описание типов данных, предусмотренных в данном языке представлено в таблице 1.2. Переменные целочисленного типа и указатели на строки находятся в стеке.

Таблица 1.2 – Типы данных языка РАА-2022

Тип данных	Описание типа данных
Строковый тип данных string	Фундаментальный тип данных. Предусмотрен для объявления строк (1 символ – 1 байт). Максимальное количество символов в строке – 255. Автоматическая инициализация строкой нулевой длины. Возможные операции: = – бинарный, присваивание значения.

Продолжение таблицы 1.2

Тип данных	Описание типа данных
Целочисленный беззнаковый тип данных uint	Фундаментальный тип данных. Предусмотрен для объявления целочисленных беззнаковых данных (4 байта). Предназначен для арифметических операций над числами. Диапазон значений от 0 до 2147483648. Автоматически инициализируется нулевым значением. Возможные операции: + (бинарный) - суммирование; - (бинарный) - вычитание; * (бинарный) - умножение; / (бинарный) - деление; = (бинарный) - присваивание значения. В качестве операторов условия условного оператора поддерживаются следующие операторы: > (бинарный) – оператор “больше”; < (бинарный) – оператор “меньше”; != (бинарный) – оператор “не равно”; >= (бинарный) – оператор “больше либо равно”; <= (бинарный) – оператор “меньше либо равно” ; == (бинарный) – оператор “сравнение”.

1.6 Преобразование типов данных

В языке программирования PAA-2022 преобразование типов данных не поддерживается, т.е. язык является строго типизированным.

1.7 Идентификаторы

Общее количество идентификаторов ограниченно максимальным размером таблицы идентификаторов (4096). Идентификаторы могут содержать символы как нижнего регистра, так и верхнего. Максимальная длина идентификатора равна 10 символам. Данные правила действуют для всех идентификаторов. Зарезервированные идентификаторы не предусмотрены. Идентификаторы не должны совпадать с ключевыми словами. Примеры идентификаторов представлены в таблице 1.3.

<буква> ::= a | b | c | d | e | f | g | h | i | j | k | l | m | n | o | p | q | r | s | t | u | v | w | x | y | z | A | B | C | D | E | F | G | H | I | J | K | L | M | N | O | P | Q | R | S | T | U | V | W | X | Y | Z

<идентификатор> ::= <буква> |<буква><идентификатор>

Таблица 1.3 – Пример идентификаторов

Идентификатор	Пример
Корректные	test Numb
Некорректные	Identifikator uint

1.8 Литералы

С помощью литералов осуществляется инициализация переменных. Все литералы являются rvalue. Имеются целочисленные литералы десятичного и восьмеричного представления, а также строковые литералы. Подробное описание литералов языка РАА-2022 представлены в таблице 1.4.

Таблица 1.4 – Описание литералов

Литералы	Пояснение	Пример
Строковые литералы	Набор символов алфавита языка, заключенных в двойные кавычки.	declare string str; str = '\'Hello world'\'; '\'Hello world'\' – строковый литерал.
Целочисленные литералы в восьмеричном представлении	Последовательность цифр 0…7 с предшествующим знаком минус или без него (знак минус не отделяется пробелом), последующим символом “o”.	new uint ch; ch = 7o; 7o – целочисленный литерал в восьмеричном представлении.
Целочисленные литералы в десятичном представлении	Последовательность цифр 0…9 с предшествующим знаком минус или без него (знак минус не отделяется пробелом).	new uint ch; ch = 15; 15 – целочисленный литерал в десятичном представлении.

Ограничения на целочисленные литералы: не могут начинаться с 0, если их значение не 0; если литерал отрицательный, после знака “-” не может идти 0.

1.9 Объявление данных

Область видимости идентификаторов «сверху вниз» (по принципу C++). В языке РАА-2022 требуется обязательное объявление переменной перед её использованием. Все переменные должны находиться внутри программного блока языка. Имеется возможность объявления одинаковых переменных в разных блоках.

1.10 Инициализация данных

При объявлении переменной допускается инициализация данных. При этом переменной будет присвоено значение литерала или идентификатора, стоящего справа от знака равенства. Объектами-инициализаторами могут быть только идентификаторы и литералы. При объявлении переменные инициализируются значением по умолчанию. Для uint значение 0, для string строка нулевой длины ('').

1.11 Инструкции языка

Все возможные инструкции языка программирования PAA-2022 представлены в общем виде в таблице 1.5.

Таблица 1.5 – Инструкции языка программирования PAA-2022

Инструкция	Запись на языке PAA-2022
Объявление переменной	new <тип данных> <идентификатор>;
Точка входа	main { … }
Объявление внешней функции	<тип данных> function <идентификатор> (<тип данных> <идентификатор>, …) {…}
Присваивание	<идентификатор> = <выражение>; Выражением может быть идентификатор, литерал, или вызов функции соответствующего типа. Для целочисленного типа выражение может быть дополнено арифметическими операциями с любым количеством операндов с использованием скобок.
Возврат значения из подпрограммы	return <идентификатор> | <литерал>;
Вывод данных	output (<идентификатор> | <литерал>);

Условный оператор	if (<условие> | (<идентификатор> | <литерал>)
	{
	…
	}
	else
	{
	…
	}
	Блок else не обязателен.

1.12 Операции языка

Языком программирования PAA-2022 предусмотрены следующие операции, представленные в таблице 1.6.

Таблица 1.6 – Приоритетности операций языка программирования PAA-2022

Операция	Приоритетность операции
( )	1
* / %	2
+ -	3
== !=	4

Продолжение таблицы 1.6

Операция	Приоритетность операции
< > <= >=	5
=	6
,	7

Максимальным значением приоритетности является “1”, минимальным “7” соответственно.

1.13 Выражения и их вычисления

Вычисление выражений – одна из важнейших задач языков программирования. В выражении используются стандартные операции: равенство, неравенство, меньше, больше, меньше или равно, больше или равно. Всякое выражение составляется согласно следующим правилам:

– Допускается использовать скобки для смены приоритета операций;

– Выражение записывается в строку без переносов;

– Использование двух подряд идущих операторов не допускается;

– Допускается использовать в выражении вызов функции, вычисляющей и возвращающей целочисленное значение;

– Операнды в арифметическом выражении не могут быть разных типов.

Перед генерацией кода каждое выражение приводится к записи в польской записи для удобства дальнейшего вычисления выражения на языке ассемблера. Преобразование выражений приведено в главе 5.

1.14 Конструкции языка

Ключевые программные конструкции языка программирования PAA-2022 представлены в таблице 1.7.

Таблица 1.7 – Программные конструкции языка PAA-2022

Конструкция	Описание

Главная функция (точка входа в приложение)	main
	{
	…
	output <имя переменной/литерал>;
	}

Функция	<тип> function <идентификатор>(<тип> <идентификатор>)
	{
	…
	output <имя переменной/литерал>;
	}

Блок	{ … }

Объявление функции допустимо только перед точкой входа в программу, так как иначе функции не будут входить в область видимости программы.

1.15 Область видимости идентификаторов

В языке PAA-2022 переменные обязаны находиться внутри программного блока функций (по принципу C++). Объявление глобальных переменных не предусмотрено. Объявление пользовательских областей видимости не предусмотрено.

1.16 Семантические проверки

Таблица с перечнем семантических проверок, предусмотренных языком PAA-2022, приведена в таблице 1.8. Часть семантических проверок выполняется на этапе лексического анализа.

Таблица 1.8 – Семантические проверки

Номер	Правило
1	Вызов функции должен соответствовать её прототипу
2	Идентификатор должен быть объявлен до его использования
3	Операнды в арифметическом выражении не могут быть разных типов
4	Каждый идентификатор может быть объявлен только один раз
5	Проверка на превышение максимального размера строкового и целочисленного литералов
6	Соответствие типа возвращаемого значения с типом функции
7	Соответствие типов в выражениях

1.17 Распределение оперативной памяти на этапе выполнения

Переменные целочисленного типа находятся в стеке, так же в стеке находятся указатели на строки. Распределение оперативной памяти происходит на этапе генерации. Промежуточный код, таблица лексем и таблица идентификаторов сохраняются в структуры с выделенной под них динамической памятью, которая очищается по окончанию работы транслятора.

1.18 Стандартная библиотека и её состав

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

Таблица 1.9 – Состав стандартной библиотеки

Имя функции	Возвращаемое значение	Принимаемые параметры	Описание
compare	uint	string x – строка string y – строка	Функция лексикографически сравнивает строку x со строкой y
strln	uint	string x - строка	Функция вычисляет длину строки x
outInt	0	uint x - число	Функция выводит на консоль число x

Продолжение таблицы 1.9

Имя функции	Возвращаемое значение	Принимаемые параметры	Описание
outStr	0	string x - строка	Функция выводит на консоль строку x

1.19 Ввод и вывод данных

Ввод данных не поддерживается языком программирования PAA-2022. output (<идентификатор или литерал>); – вывод в стандартный поток вывода. В зависимости от типа параметра определяется функция: outStr или outInt, которые входят в состав стандартной библиотеки и описаны в таблице 1.9.

1.20 Точка входа

В языке PAA-2022 каждая программа должна содержать главную функцию (точку входа) main, с первой инструкции которой начинается последовательное выполнение команд программы.

1.21 Препроцессор

Препроцессор в языке программирования PAA-2022 не предусмотрен.

1.22 Соглашения о вызовах

Соглашение о вызовах – это правила передачи управления от вызывающего к вызываемому коду, определяющие способы передачи параметров и результата вычислений, возврат в точку вызова.

В языке PAA-2022 вызов функций происходит по соглашению о вызовах stdcall. Особенности stdcall:

– все параметры функции передаются через стек;

– память освобождает вызываемый код;

– занесение в стек параметров идёт справа налево.

1.23 Объектный код

Язык программирования PAA-2022 транслируется в язык ассемблера, а затем в объектный код.

1.24 Классификация сообщений транслятора

В случае возникновения ошибки в коде программы на языке PAA-2022 и выявления её транслятором в текущий файл протокола выводится сообщение. Классификация сообщений ошибок приведена в таблице 1.10.

Таблица 1.10 – Классификация сообщений транслятора

Интервал	Описание ошибок
0-99	Системные ошибки
100-109	Ошибки параметров
110-119	Ошибки открытия и чтения файлов

Продолжение таблицы 1.10

Интервал	Описание ошибок
120-200	Ошибки лексического анализа
300-399	Ошибки семантического анализа
600-699	Ошибки синтаксического анализа

1.25 Контрольный пример

Контрольный пример представлен в приложении А.

2 Структура транслятора

2.1 Компоненты транслятора, их назначение и принципы взаимодействия

Транслятор преобразует программу, написанную на языке PAA-2022 в программу на языке ассемблера. Компонентами транслятора являются лексический, синтаксический и семантический анализаторы, а также генератор кода на язык ассемблера. Принцип их взаимодействия представлен на рисунке 2.1.

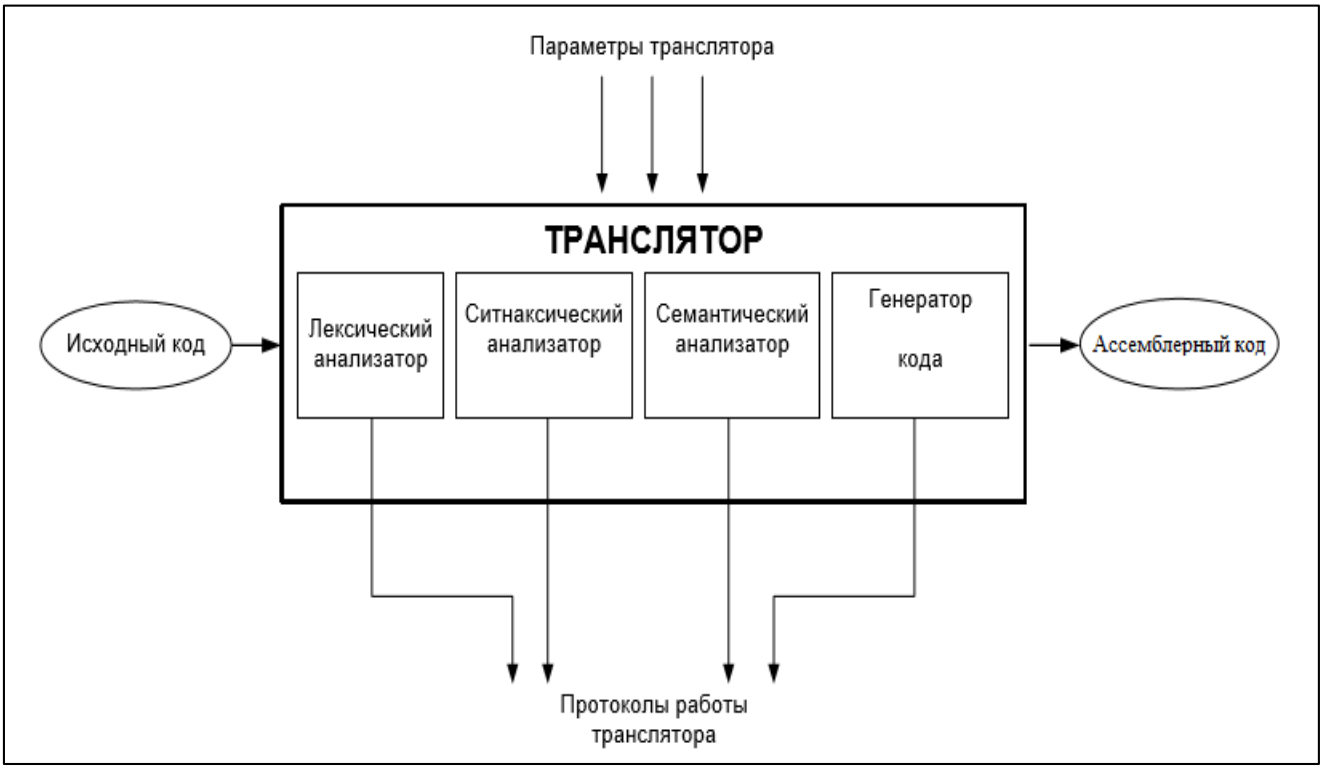

Рисунок 2.1 – Структура транслятора

Лексический анализ – первая фаза трансляции. Назначением лексического анализатора является нахождение ошибок лексики языка и формирование таблицы лексем и таблицы идентификаторов. Подробнее описан в главе 3.

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. Подробное описание представлено в главе 5.

В трансляторе, созданном для языка PAA-2022, часть функции семантического анализатора возложена на лексический анализатор.

Синтаксический анализ – это основная часть транслятора, предназначенная для распознавания синтаксических конструкций и формирования промежуточного кода. Входным параметром для синтаксического анализа является таблица лексем. Синтаксический анализатор распознаёт синтаксические конструкции, выявляет синтаксические ошибки при их наличии и формирует дерево разбора. Подробнее рассмотрен в главе 4.

Генератор кода – этап транслятора, выполняющий генерацию ассемблерного кода на основе полученных данных на предыдущих этапах трансляции.

Генератор кода принимает на вход таблицы идентификаторов и лексем и транслирует код на языке РАА-2022, прошедший все предыдущие этапы, в код на языке Ассемблера. Более полно описан в главе 7.

2.2 Перечень входных параметров транслятора

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

Таблица 2.1 – Входные параметры транслятора языка РАА-2022

Входной параметр	Описание	Значение по умолчанию
-in:<путь к in-файлу>	Входной файл с расширением .txt, в котором содержится исходный код на РАА-2022.	Не предусмотрено
-log:<путь к log-файлу>	Файл журнала для вывода протоколов работы программы.	<имя in-файла>.log
-out:<имя_файла>	Выходной файл – результат работы транслятора. Содержит исходный код на языке ассемблера.	<имя in-файла>.asm
-mfst:<имя_файла>	Файл для записи результата синтаксического разбора.	<имя in-файла>.mfst

2.3 Протоколы, формируемые транслятором

Таблица с перечнем протоколов, формируемых транслятором языка РАА-2022 и их назначением представлена в таблице 2.2

Таблица 2.2 – Протоколы, формируемые транслятором языка РАА-2022

Формируемый протокол	Описание протокола
Файл журнала с параметром “-log:”	Содержит информацию о входных параметрах в приложение, о этапе проверки символов на допустимость, а также результат работы лексического анализатора.
Выходной файл с параметром “-out:”	Содержит сгенерированный код на языке ассемблера.
Выходной файл с параметром “-mfst:”	Результат работы синтаксического анализа. Содержит правила разбора, а также трассировку.

3 Разработка лексического анализатора

3.1 Структура лексического анализатора

Лексический анализатор – часть транслятора, выполняющая лексический анализ. Лексический анализатор принимает обработанный и разбитый на отдельные компоненты исходный код на языке PAA-2022. На выходе формируется таблица лексем и таблица идентификаторов. Структура лексического анализатора представлена на рисунке 3.1.

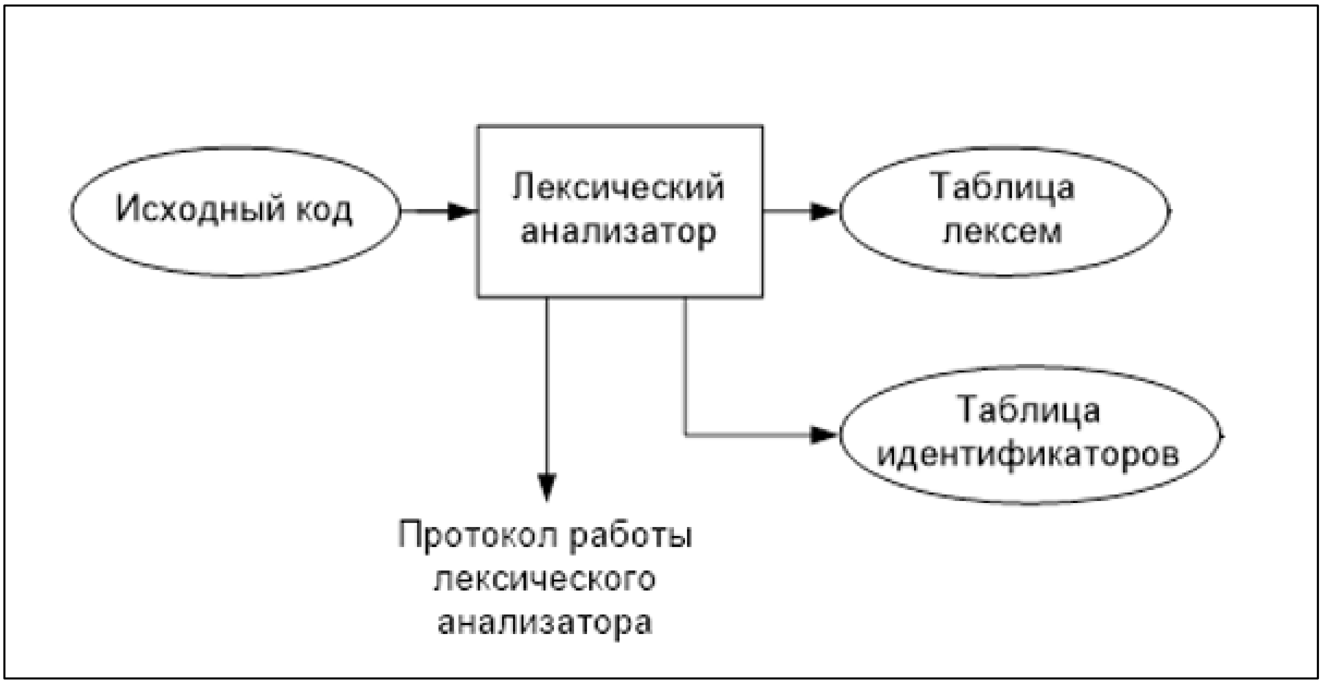

Рисунок 3.1 – Структура лексического анализатора PAA-2022

3.2 Контроль входных символов

Таблица для контроля входных символов представлена на рисунке 3.2.

```
#define IN_CODE_TABLE {\
    IN::F,   IN::F,  IN::F, IN::F, IN::F, IN::F,  IN::F, IN::F, IN::F,  IN::SPC, IN::S,  IN::F,  IN::F,   IN::F,  IN::F,  IN::F, \
    IN::F,   IN::F,  IN::F, IN::F, IN::F, IN::F,  IN::F, IN::F, IN::F,  IN::F,   IN::F,  IN::F,  IN::F,   IN::F,  IN::F,  IN::F, \
    IN::SPC, IN::LX, IN::F, IN::F, IN::F, IN::LX, IN::F, IN::T, IN::LX, IN::LX,  IN::LX, IN::LX, IN::LX,  IN::LX, IN::F,  IN::LX,\
    IN::T,   IN::T,  IN::T, IN::T, IN::T, IN::T,  IN::T, IN::T, IN::T,  IN::T,   IN::T,  IN::LX, IN::LX,  IN::LX, IN::LX, IN::F, \
    IN::F,   IN::T,  IN::T, IN::T, IN::T, IN::T,  IN::T, IN::T, IN::T,  IN::T,   IN::T,  IN::T,  IN::T,   IN::T,  IN::T,  IN::T, \
    IN::T,   IN::T,  IN::T, IN::T, IN::T, IN::T,  IN::T, IN::T, IN::T,  IN::T,   IN::T,  IN::F,  IN::F,   IN::F,  IN::F,  IN::F, \
    IN::F,   IN::T,  IN::T, IN::T, IN::T, IN::T,  IN::T, IN::T, IN::T,  IN::T,   IN::T,  IN::T,  IN::T,   IN::T,  IN::T,  IN::T, \
    IN::T,   IN::T,  IN::T, IN::T, IN::T, IN::T,  IN::T, IN::T, IN::T,  IN::T,   IN::T,  IN::LX, IN::F,   IN::LX, IN::F,  IN::F, \
                                                                                                                  \
    IN::F,   IN::F,  IN::F, IN::F, IN::F, IN::F,  IN::F, IN::F, IN::F,  IN::F,   IN::F,  IN::F,  IN::F,   IN::F,  IN::F,  IN::F, \
    IN::F,   IN::F,  IN::F, IN::F, IN::F, IN::F,  IN::F, IN::F, IN::F,  IN::F,   IN::F,  IN::F,  IN::F,   IN::F,  IN::F,  IN::F, \
    IN::F,   IN::F,  IN::F, IN::F, IN::F, IN::F,  IN::F, IN::F, IN::F,  IN::F,   IN::F,  IN::F,  IN::F,   IN::F,  IN::F,  IN::F, \
    IN::F,   IN::F,  IN::F, IN::F, IN::F, IN::F,  IN::F, IN::F, IN::F,  IN::F,   IN::F,  IN::F,  IN::F,   IN::F,  IN::F,  IN::F, \
    IN::T,   IN::T,  IN::T, IN::T, IN::T, IN::T,  IN::T, IN::T, IN::T,  IN::T,   IN::T,  IN::T,  IN::T,   IN::T,  IN::T,  IN::T, \
    IN::T,   IN::T,  IN::T, IN::T, IN::T, IN::T,  IN::T, IN::T, IN::T,  IN::T,   IN::T,  IN::T,  IN::T,   IN::T,  IN::T,  IN::T, \
    IN::T,   IN::T,  IN::T, IN::T, IN::T, IN::T,  IN::T, IN::T, IN::T,  IN::T,   IN::T,  IN::T,  IN::T,   IN::T,  IN::T,  IN::T, \
    IN::T,   IN::T,  IN::T, IN::T, IN::T, IN::T,  IN::T, IN::T, IN::T,  IN::T,   IN::T,  IN::T,  IN::T,   IN::T,  IN::T,  IN::T, \
}
```

Рисунок 3.2. – Таблица контроля входных символов

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

Описание значения символов: T – разрешённый символ, F – запрещённый символ, I – игнорируемый символ, SPC – разделитель лексем, S – символ переноса строки, LX – символ литерала.

3.3 Удаление избыточных символов

Избыточными символами являются символы табуляции, пробелы и переходы на новую строку.

Избыточные символы удаляются на этапе разбиения исходного кода на токены.

Описание алгоритма удаления избыточных символов:

1) Посимвольно считываем файл с исходным кодом программы.

2) Встреча пробела или знака табуляции является своего рода встречей символа-сепаратора.

3) В отличие от других символов-сепараторов не записываем в очередь лексем эти символы, т.е. игнорируем.

3.4 Перечень ключевых слов

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

Таблица 3.1 Соответствие ключевых слов, символов операций и сепараторов с лексемами

Токен	Лексема	Пояснение
uint, string	t	Названия типов данных языка.
Идентификатор	i	Длина идентификатора – 10 символов.
литерал	l	Литерал любого доступного типа.
function	f	Объявление функции.
return	r	Выход из функции/процедуры.
main	m	Главная функция.
new	n	Объявление переменной.
output	p	Вывод данных.
if	e	Указывает на начало условного блока.
else	c	Указывает на ложную ветвь условного блока.
;	;	Разделение выражений.
,	,	Разделение параметров функций.
{	}	Начало блока/тела функции.
}	{	Закрытие блока/тела функции.
(	(	Передача параметров в функцию, приоритет операций.
)	)	Закрытие блока для передачи параметров, приоритет операций.
=	=	Знак присваивания

Продолжение таблицы 3.1

> < <= >= == !=	> < $ # ~ &	Знаки логических операторов
+ – * /	+ – * /	Знаки операций

Каждому выражению соответствует регулярное выражение, по которому происходит разбор данного выражения. На каждое регулярное выражение в массиве подаётся фраза и с помощью метода std::regex_match, стандартной библиотеки <regex>, соответствующего данному регулярному выражению, происходит разбор. В случае успешного разбора выражения записывается в таблицу лексем. Если выражение является идентификатором или литералом, информация также заносится в таблицу идентификаторов. В приложении А находятся регулярные выражения, соответствующие лексемам языка PAA-2022.

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 идентификатором (idxTI). Таблица идентификаторов содержит имя идентификатора (id), область видимости идентификатора (scope), номер в таблице лексем (idxfirstLE), тип данных (iddatatype), тип идентификатора (idtype) и значение (value). Код C++ со структурой таблицы лексем представлен на листинге 3.3. Код C++ со структурой таблицы идентификаторов представлен на листинге 3.4.

```
struct Entry {
	char lexema;
	int sn;
	int idxTI;
};
struct LexTable {
	int maxsize;
	int size;
	Entry* table;
};
```

Листинг 3.3 – Код структуры таблицы лексем

```
enum class IDDATATYPE { UINT = 1, STR = 2, BOOL = 3 };
enum class IDTYPE { V = 1, F = 2, P = 3, L = 4 };

struct Entry {
	int			idxfirstLE;
	std::string id;
```

```
	std::string scope;
	IDDATATYPE	iddatatype;
	IDTYPE		idtype;
	union
	{
		int vint;
		struct {
			char str[TI_STR_MAXSIZE - 1];
			int len;
		} vstr;
	} value;
};

struct IdTable {
	int maxsize;
	int size;
	Entry* table;
};
```

Листинг 3.4 – Код структуры таблицы идентификаторов

3.6 Структура и перечень сообщений лексического анализатора

Перечень сообщений лексического анализатора представлен на листинге 3.4.

```
	ERROR_ENTRY(120, "[Лексическая] Превышен максимальный размер
таблицы лексем"),
	ERROR_ENTRY(121, "[Лексическая] Таблица лексем переполнена"),
	ERROR_ENTRY(122, "[Лексическая] Нераспознаянная лексема"),
	ERROR_ENTRY(123, "[Лексическая] Выход за пределы таблицы лексем"),
	ERROR_ENTRY(124, "[Лексическая] Превышена длина строковго
литерала"),
	ERROR_ENTRY(125, "[Лексическая] Превышен максимальный размер
таблицы идентификаторов"),
	ERROR_ENTRY(126, "[Лексическая] Таблица идентификаторов
переполнена"),
	ERROR_ENTRY(127, "[Лексическая] Выход за пределы таблицы
идентификаторов"),
	ERROR_ENTRY(128, "[Лексическая] Повторное объявление main"),
	ERROR_ENTRY(129, "[Лексическая] Отсутствие входной точки программы
функции main"),
	ERROR_ENTRY(130, "[Лексическая] Превышено значение целочисленного
литерала"),
	ERROR_ENTRY(131, "[Лексическая] Необъявленный идентификатор"),
	ERROR_ENTRY(132, "[Лексическая] Незакрытый строковый литерал"),
	ERROR_ENTRY(133, "[Лексическая] Превышена длина идентификатора"),
```

Листинг 3.4 – Перечень ошибок лексического анализатора

3.7 Принцип обработки ошибок

При возникновении ошибки транслятор завершает свою работу. Для обработки ошибок лексический анализатор использует таблицу с сообщениями.

Структура сообщений содержит информацию о номере сообщения, номер строки и позицию, где было вызвано сообщение в исходном коде, информацию об ошибке.

3.8 Параметры лексического анализатора

Лексический анализатор принимает обработанный и разбитый на отдельные компоненты исходный код на языке PAA-2022. На выходе формируется таблица лексем и таблица идентификаторов.

Результаты работы лексического анализатора, а именно таблицы лексем и идентификаторов выводятся в файл журнала.

3.9 Алгоритм лексического анализа

Лексический анализ выполняется программой (входящей в состав транслятора), называемой лексическим анализатором. Цель лексического анализа — выделение и классификация лексем в тексте исходной программы. Лексический анализатор распознаёт и разбирает цепочки исходного текста программы. Этот разбор основывается на работе конечных автоматов, которую можно представить в виде графов.

Регулярные выражения — аналитический или формульный способ задания регулярных языков. Они состоят из констант и операторов, которые определяют множества строк и множество операций над ними. Любое регулярное выражение можно представить в виде графа.

Пример. Регулярное выражение для ключевого слова num: `'uint'`.

Граф конечного автомата для этой лексемы представлен на рисунке 3.5. S0 – начальное состояние, S4 – конечное состояние автомата.

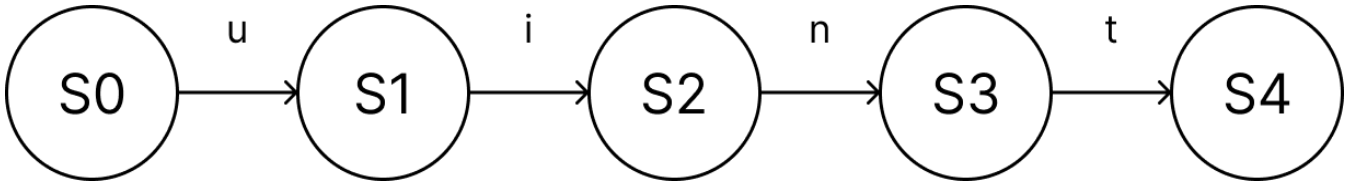

Рисунок 3.5 – Граф переходов для цепочки 'uint'

3.10 Контрольный пример

Результат работы лексического анализатора – таблицы лексем и идентификаторов – представлен в приложении А в листингах 2 и 3.

4 Разработка синтаксического анализатора

4.1 Структура синтаксического анализатора

Синтаксический анализ – это фаза трансляции, выполняемая после лексического анализа и предназначенная для распознавания синтаксических конструкций. Входом для синтаксического анализа является таблица лексем и таблица идентификаторов, полученные после фазы лексического анализа. Выходом – дерево разбора.

Структура синтаксического анализатора представлена на рисунке 4.1.

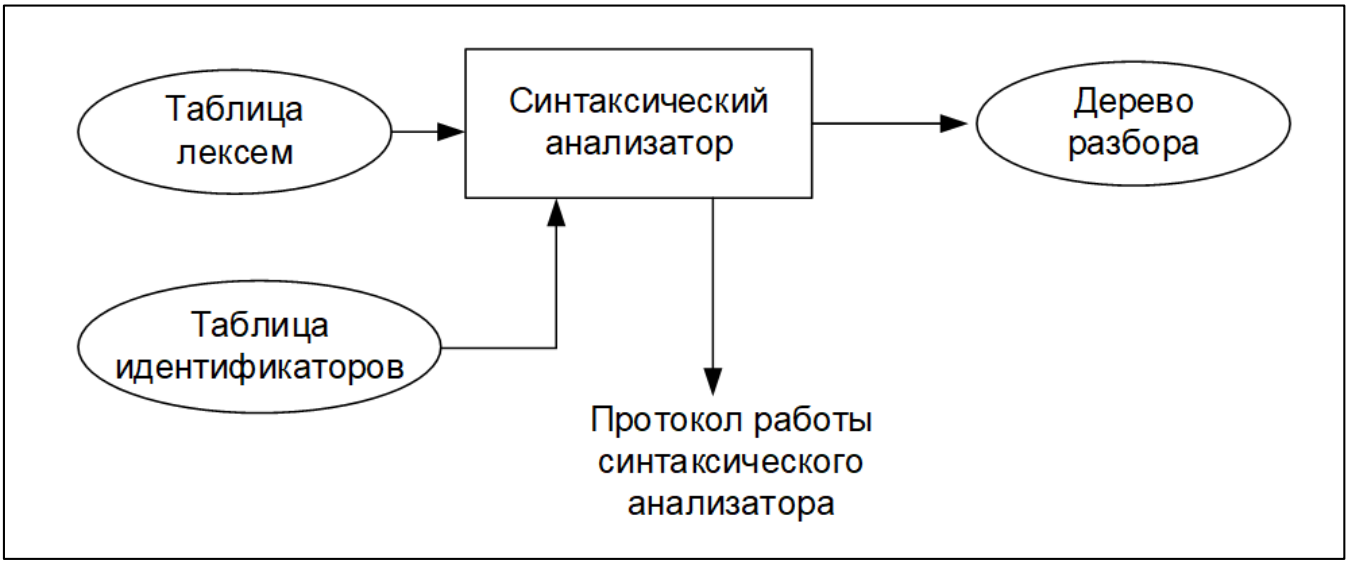

Рисунок 4.1 – Структура синтаксического анализатора

4.2 Контекстно свободная грамматика, описывающая синтаксис языка

В синтаксическом анализаторе транслятора языка РАА-2022 используется контекстно-свободная грамматика типа II в иерархии Хомского (Контекстно-свободная грамматика) $G = \langle T, N, P, S \rangle$, где:

T – множество терминальных символов (было описано в разделе 1.2);

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

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

S – начальный символ грамматики, являющийся нетерминалом.

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

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

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

Грамматика языка РАА-2022 представлена в приложении Б.

TS – терминальные символы, которыми являются сепараторы, знаки арифметических операций и некоторые строчные буквы.

NS – нетерминальные символы, представленные несколькими заглавными буквами латинского алфавита.

Таблица 4.1 – Перечень правил, составляющих грамматику языка и описание нетерминальных символов PAA-2022

Нетерминал	Цепочки правил	Какие правила порождает
S	S->tfiPTS S->mT S->dtfiP;S	Стартовые правила, описывающие общую структуру программы
T	T->{rV;} T->{KrV;}	Привила для тела функций
P	P->(E) P->()	Правила для параметров объявляемых функций
E	E->ti E->ti, E	Правила для списка параметров функции
F	F->(N) F->()	Правила для вызова функций
N	N->i N->l N->i, N N->l, N	Правила для параметров вызываемой функции
Z	Z->(iLi) Z->(iLl) Z->(lLi) Z->(lLl) Z->(V)	Правила для условного оператора
L	L->< L->>>> L->$ L-># L->~ L->&	Правила для логических операторов
A	A->+ A->- A->* A->/ A->%	Правила для арифметических операторов
V	V->i V->l	Правила для простых выражений
W	W->l W->i W->(W) W->(W)AW W->iF W->iAW	Правила для сложных выражений

Продолжение таблицы 4.1

Нетерминал	Цепочки правил	Какие правила порождает
W	W->l W->i W->(W) W->(W)AW W->iF W->iAW	Правила для сложных выражений
K	K->dti=W;K K->i=W; K->dti;K K->i=W;K K->pV;K K->{K} K->{K}K K->eZ{K}K K->eZ{K}y{K}K K->iF;K K->dti=W; K->dti; K->pV; K->eZ{K} K->eZ{K}y{K} K->iF;	Правила для синтаксических конструкций

4.3 Построение конечного магазинного автомата

Конечный автомат с магазинной памятью представляет собой семерку $M = \langle Q, V, Z, \delta, q_0, z_0, F \rangle$, описание которой представлено в таблице 4.2. Структура данного автомата показана в приложении В.

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 является множеством терминальных и нетерминальных символов, описание которых содержится в разделе 1.2 и в таблице 4.1.
Z	Алфавит специальных магазинных символов	Алфавит магазинных символов содержит стартовый символ и маркер дна стека.

Продолжение таблицы 4.2

Компонента	Определение	Описание
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 (нетерминальный символ S).
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 включают в себя структуру магазинного автомата, выполняющего разбор исходной ленты, и структуры грамматики Грейбах, описывающей синтаксические правила языка PAA-2022. Данные структуры представлены в приложении В.

4.5 Описание алгоритма синтаксического разбора

Принцип работы автомата следующий:

1) В магазин записывается стартовый символ грамматики;
2) На основе полученных ранее таблиц формируется входная лента;
3) Запускается автомат;
4) Выбирается цепочка, соответствующая нетерминальному символу, записывается в магазин в обратном порядке;
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 не терминала;
6) Если в магазине встретился не терминал, 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 листинге 4.1.

```
        ERROR_ENTRY(600, "[Синтаксическая] Неверная структура программы"),
        ERROR_ENTRY(601, "[Синтаксическая] Не найден список параметров
функции"),
        ERROR_ENTRY(602, "[Синтаксическая] Ошибка в теле функции"),
        ERROR_ENTRY(603, "[Синтаксическая] Ошибка в теле процедуры"),
        ERROR_ENTRY(604, "[Синтаксическая] Ошибка в списке параметров
функции"),
        ERROR_ENTRY(605, "[Синтаксическая] Ошибка в вызове
функции/выражении"),
        ERROR_ENTRY(606, "[Синтаксическая] Ошибка в списке фактических
параметров функции"),
        ERROR_ENTRY(607, "[Синтаксическая] Ошибка в условии условного
выражения"),
        ERROR_ENTRY(608, "[Синтаксическая] Неверный условный оператор"),
        ERROR_ENTRY(609, "[Синтаксическая] Неверный арифметический
оператор"),
        ERROR_ENTRY(610, "[Синтаксическая] Неверное выражение. Ожидаются
только идентификаторы/литералы"),
        ERROR_ENTRY(611, "[Синтаксическая] Ошибка в арифметическом
выражении"),
        ERROR_ENTRY(612, "[Синтаксическая] Недопустимая синтаксическая
конструкция"),
        ERROR_ENTRY(613, "[Синтаксическая] Синтаксический анализ завершён с
ошибкой"),
```

Листинг 4.1 – Перечень сообщений синтаксического анализатора

4.7 Параметры синтаксического анализатора и режимы его работы

Входным параметром синтаксического анализатора является таблица лексем, полученная на этапе лексического анализа, а также правила контекстно-свободной грамматики в форме Грейбах.

Выходными параметрами являются трассировка прохода таблицы лексем (при наличии разрешающего ключа) и правила разбора, которые записываются в файл протокола данного этапа обработки.

4.8 Принцип обработки ошибок

Обработка ошибок происходит следующим образом:

1) Синтаксический анализатор перебирает все правила и цепочки правила грамматики для нахождения подходящего соответствия с конструкцией, представленной в таблице лексем.

2) Если невозможно подобрать подходящую цепочку, то генерируется соответствующая ошибка.

3) Все ошибки записываются в общую структуру ошибок.

4) В случае нахождения ошибки, после всей процедуры трассировки в протокол будет выведено диагностическое сообщение.

В структуре грамматики Грейбах цепочки в правилах расположены в порядке приоритета, самые часто используемые располагаются выше, а те, что используются реже – ниже.

4.9 Контрольный пример

Пример разбора синтаксическим анализатором исходного кода на языке РАА-2022 представлен в приложении Г. Дерево разбора исходного кода также представлено в приложении Г.

5 Разработка семантического анализатора

5.1 Структура семантического анализатора

Семантический анализатор принимает на свой вход результаты работ лексического и синтаксического анализаторов, то есть таблицы лексем, идентификаторов и результат работы синтаксического анализатора, то есть дерево разбора, и последовательно ищет необходимые ошибки. Некоторые проверки (такие как проверка на единственность точки входа, проверка на предварительное объявление переменной) осуществляются в процессе лексического анализа. Общая структура обособленно работающего (не параллельно с лексическим анализом) семантического анализатора представлена на рисунке 5.1.

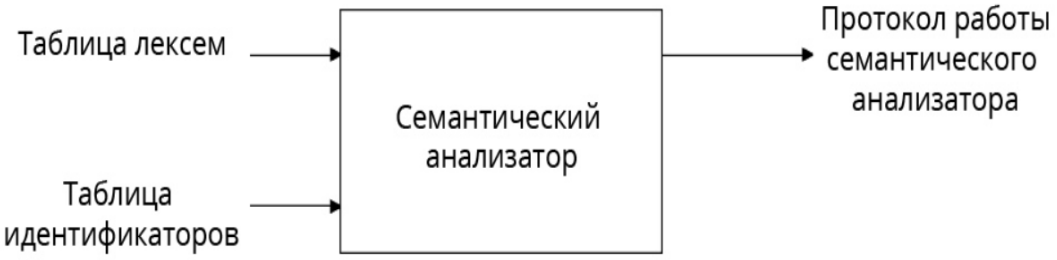

Рисунок 5.1 – Общая структура обособленно работающего семантического анализатора

5.2 Функции семантического анализатора

Семантический анализатор проверяет правильность составления программных конструкций. При невозможности подобрать правило перехода будет выведен код ошибки. Информация об ошибках выводится в консоль, а также в протокол работы.

Все функции семантического анализатора, осуществляющие проверку лексем, принимают индекс, под которым стоит та или иная лексема таблицы лексем, попадающая под определенное условие. На выходе же, функции ничего не возвращают в случае успешной проверки и генерируют ошибку в случае ошибки.

Таблица 5.1 – Функции семантического анализатора, реализующие проверку

Функция	Краткое описание
void checkTypeOfReturn(int& i)	Проверка возвращаемого значения
void checkCallOfFunction(int& i)	Проверка принимаемых параметров
void checkExpressioin(int& i)	Проверка выражений на корректность
void checkIfExpression(int& i)	Проверка условия в условном операторе

5.3 Структура и перечень сообщений семантического анализатора

Сообщения, формируемые семантическим анализатором, представлены на листинге 5.1.

```
	ERROR_ENTRY(300, "[Семантическая] Функция main должна возвращать
числовое значение"),
	ERROR_ENTRY(301, "[Семантическая] Тип функции и тип возвращаемого
значения отличаются"),
	ERROR_ENTRY(302, "[Семантическая] Несоответствие типов параметров в
вызываемой функции"),
	ERROR_ENTRY(303, "[Семантическая] Неверное количество параметров
вызываемой функции"),
	ERROR_ENTRY(304, "[Семантическая] Несоответствие присваиваемого
типа данных"),
	ERROR_ENTRY(305, "[Семантическая] Использование имени переменной в
качестве функции"),
	ERROR_ENTRY(306, "[Семантическая] Попытка переопределения
идентификатора"),
	ERROR_ENTRY(307, "[Семантическая] Недопустимое значение переменной
типа UINT"),
	ERROR_ENTRY(308, "[Семантическая] Недопустимая операция над данным
типом"),
	ERROR_ENTRY(309, "[Семантическая] Неверное условие в условном
операторе"),
	ERROR_ENTRY(310, "[Семантическая] Недопустимый возврат функции типы
UINT"),
```

Листинг 5.1 – Перечень сообщений семантического анализатора

5.4 Принцип обработки ошибок

При обнаружении ошибки в исходном коде программы семантический анализатор формирует сообщение об ошибке и выводит его в файл с протоколом работы, заданный параметром – log:.

5.5 Контрольный пример

Результат работы контрольного примера расположен в приложении А, где показан результат лексического анализатора, т.к. представленные таблицы лексем и идентификаторов проходят лексическую и часть семантических проверок одновременно.

В таблице 5.2 приведен пример ошибок, диагностируемых семантическим анализатором на разных этапах трансляции.

Таблица 5.2 – Пример ошибок, диагностируемых семантическим анализатором

Исходный код	Диагностическое сообщение
new uint x = strLen ('hello', 'world');	Ошибка: 303 [Семантическая] Неверное количество параметров вызываемой функции Строка: 41

main
{
return 'test';
}
	Ошибка: 300 [Семантическая] Функция main должна возвращать числовое значение
	Строка: 46

Продолжение таблицы 5.2

if (x)
{
ret = 1;
}
	Ошибка: 309 [Семантическая]
	Неверное условие в условном операторе
	Строка: 8

6 Преобразование выражений

6.1 Выражения, допускаемые языком

В языке РАА-2022 допускаются выражения, применимые к целочисленным типам данных. В выражениях поддерживаются арифметические операции, такие как +, -, *, /, %, (), и вызовы функций как операнды арифметических выражений. Приоритет операций представлен в таблице 6.1.

Таблица 6.1 – Приоритет операций в языке РАА-2022

Приоритет	Операция
0	(
0	)
1	,
2	+
2	-
3	*
3	/
3	%

Скобки, в зависимости от их применения, могут иметь разный приоритет, либо 0, либо 4.

6.2 Польская запись и принцип ее построения

Выражения в языке РАА-2022 преобразовываются к обратной польской записи.

Польская запись – это альтернативный способ записи арифметических выражений, преимущество которого состоит в отсутствии скобок.

Обратная польская запись – это форма записи математических и логических выражений, в которой операнды расположены перед знаками операций.

Алгоритм построения:

– исходная строка: выражение;

– результирующая строка: польская запись;

– стек: пустой;

– результирующая строка: польская запись;

– исходная строка просматривается слева направо;

– операнды переносятся в результирующую строку в порядке их следования;

– операция записывается в стек, если стек пуст или в вершине стека лежит отрывающая скобка;

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

– запятая не помещается в стек, если в стеке операции, то все выбираются в строку;

– отрывающая скобка помещается в стек;

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

– закрывающая скобка с приоритетом, равным 4, выталкивает все до открывающей с таким же приоритетом и генерирует @ (@ – специальный символ, в который записывается информация о вызываемой функции), а в поле приоритета для данной лексемы записывается число параметров вызываемой функции;

– по итогам разбора исходной строки все операции, оставшиеся в стеке, выталкиваются в результирующую строку.

Использование польской записи позволяет вычислить выражение за один проход.

Таблица 6.2 – Пример преобразования выражения в обратную польскую запись

Исходная строка	Результирующая строка	Стек
b*2 - i(l)		
*2 - i(l)	b	
2 - i(l)	b	*
- u(l)	b2	*
u(l)	b2*	-
(l)	b2*	-
l)	b2*	-
)	b2*i	-
	b2*i@$_1$-	

6.3 Программная реализация обработки выражений

Программная реализация алгоритма преобразования выражений к польской записи представлена в приложении Д.

6.4 Контрольный пример

В приложении Д приведены изменённые таблицы лексем и идентификаторов, отображающие результаты преобразования выражений в польский формат.

7 Генерация кода

7.1 Структура генератора кода

Генерация объектного кода — это перевод компилятором внутреннего представления исходной программы в цепочку символов выходного языка. На вход генератора подаются таблицы лексем и идентификаторов, на основе которых генерируется файл с ассемблерным кодом.

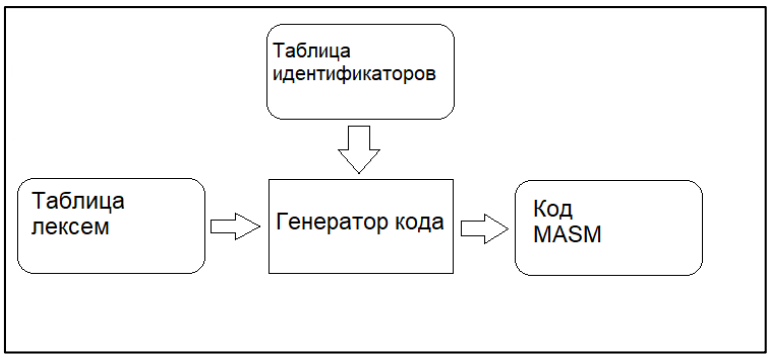

Рисунок 7.1 – Структура генератора кода

Генератор кода последовательно проходит таблицу лексем, при необходимости обращаясь к таблице идентификаторов. В зависимости от пройденных лексем выполняется генерация кода ассемблера.

7.2 Представление типов данных в оперативной памяти

Элементы таблицы идентификаторов расположены в разных сегментах языка ассемблера – .data и .const. Идентификаторы языка PAA-2022 размещены в сегменте данных (.data). Литералы – в сегменте констант (.const). Соответствия между типами данных идентификаторов на языке PAA-2022 и на языке ассемблера приведены в таблице 7.1.

Таблица 7.1 – Соответствия типов идентификаторов языка PAA-2022 и языка Ассемблера

Тип идентификатора на языке PAA-2022	Тип идентификатора на языке ассемблера	Пояснение
uint	DWORD	Хранит целочисленный беззнаковый тип данных.
string	DWORD	Хранит указатель на начало строки.

7.3 Статическая библиотека

В языке PAA-2022 предусмотрена статическая библиотека. Статическая библиотека содержит функции, написанные на языке C++. Объявление функций статической библиотеки генерируется автоматически в коде ассемблера.

Стандартная библиотека находится в директории языка и при генерации кода подключается автоматически. Путь к библиотеке генерируется автоматически на

стадии генерации кода.

Функции статической библиотеки приведены в таблице 7.3.

Таблица 7.3 – Функции статической библиотеки

Функция	Назначение
void outStr(char* str)	Вывод на консоль строки str
void outnum(unsigned long num)	Вывод на консоль целочисленной беззнаковой переменной num
unsigned long strLen(char *str)	Возвращает длину строки sty
unsigned long strCmp(char *str1, *str2)	Возвращает результат лексикографического сравнения строк str1, str2

7.4 Особенности алгоритма генерации кода

В языке РАА-2022 генерация кода строится на основе таблиц лексем и идентификаторов. Общая схема работы генератора кода представлена на рисунке 7.2.

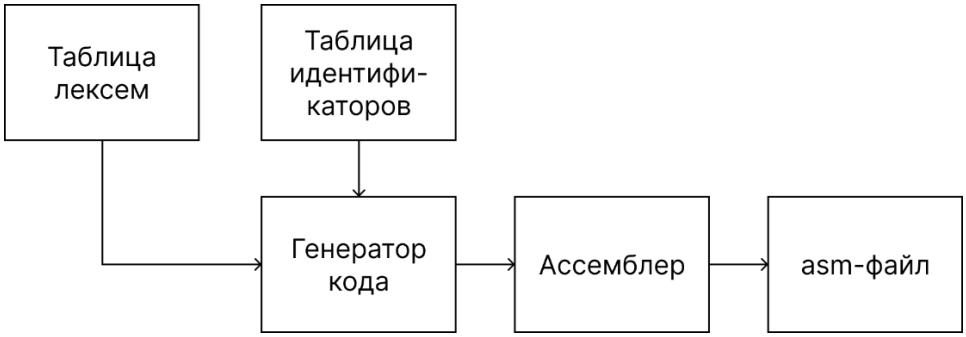

Рисунок 7.2 – Общая схема работы генератора кода

Таблица 7.4 – перечень и описание разработанных для реализации генерации кода функций.

Функция	Краткое описание
void head()	Задает “шапку” ассемблерного файла. Подключает все необходимые библиотеки, указывает модель памяти и соглашение о вызовах, указывает прототипы всех стандартных функций из библиотеки и т.д.
void constants()	Создает сегмент констант
void data()	Создает сегмент данных. Объявляет переменные
void code()	Создает сегмент кода. Управляет функциями для создания функций
void generateFunctionProto(int& i)	Создает функции и их прототипы
void generateFunctioinBody(int& i)	Создает тело функции
void generateFunctionReturn(int& i)	Создает возвращаемое значение

Продолжение таблицы 7.4

void generatePrint(int& i)	Создает функцию вывода
void generateEqual(int& i)	Создает выражения
void generateIfStatement(int& i)	Создает условные конструкции

7.5 Входные параметры генератора кода

На вход генератору кода поступают таблицы лексем и идентификаторов исходного код программы на языке PAA-2022. Результаты работы 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 также приведён в приложении Д.

8 Тестирование транслятора

8.1 Общие положения

В языке PAA-2022 не разрешается использовать запрещённые входным алфавитом символы. Результат использования запрещённого символа показан в таблице 8.1.

Таблица 8.1 – Тестирование фазы проверки на допустимость символов

Исходный код	Диагностическое сообщение

main
{
@
}
	Ошибка 111: Недопустимый символ в исходном файле (-in), строка 3 ,позиция 2.

8.2 Результаты тестирования

На этапе лексического анализа могут возникнуть ошибки, описанные в пункте 3.7.

Результаты тестирования лексического анализатора показаны в таблице 8.2.

Таблица 8.2 – Тестирование лексического анализатора

Исходный код	Диагностическое сообщение

main
{
7u
}
	Ошибка 122: [Лексическая] Нераспознанная лексема, строка 3 ,слово 7u.

main
{
x = 5;
return 0;
}
	Ошибка 131: [Лексическая] Необъявленный идентификатор, строка 3.

string function fi(uint x)
{
return ‘a’;
}
	Ошибка 129: [Лексическая] Отсутствие входной точки программы функции main.

main
{
return 0;
}
main
{
return 1;
}
	Ошибка 128: [Лексическая] Повторное объявление main.

Продолжение таблицы 8.2

Исходный код	Диагностическое сообщение

main
{
new uint abcdefghjkl;
return 0;
}
	Ошибка 133: [Лексическая] Превышена длина идентификатора, строка 3.

На этапе лексического анализа обрабатываются ошибки, которые препятствую правильному построению таблицы лексем и идентификаторов.

На этапе синтаксического анализа могут возникнуть ошибки, описанные в пункте 4.6.

Таблица 8.3 – Тестирование синтаксического анализатора

Исходный код	Диагностическое сообщение
uint function fi(uint p, 6)	604: строка 1,[Синтаксическая] Ошибка в списке параметров функции 604: строка 1,[Синтаксическая] Ошибка в списке параметров функции 601: строка 1,[Синтаксическая] Не найден список параметров функции Ошибка 613: [Синтаксическая] Синтаксический анализ завершён с ошибкой

main
{
output 5;
}
	612: строка 4,[Синтаксическая] Недопустимая синтаксическая конструкция
	610: строка 3,[Синтаксическая] Неверное выражение. Ожидаются только идентификаторы/литералы
	610: строка 3,[Синтаксическая] Неверное выражение. Ожидаются только идентификаторы/литералы
	Ошибка 613: [Синтаксическая] Синтаксический анализ завершён с ошибкой

main
{
if (5 + 7)
{
output 10;
}
}
	608: строка 3,[Синтаксическая] Неверный условный оператор
	608: строка 3,[Синтаксическая] Неверный условный оператор
	608: строка 3,[Синтаксическая] Неверный условный оператор
	Ошибка 613: [Синтаксическая] Синтаксический анализ завершён с ошибкой

Итоги тестирования семантического анализатора приведены в таблице 8.4.

Таблица 8.4 – Тестирование семантического анализатора

Исходный код	Диагностическое сообщение

main {
new uint x = 5 + 'z';
return 0;
}
	Ошибка 308: [Семантическая] Недопустимая операция над данным типом, строка 3

Продолжение таблицы 8.4

main
{
return 'z';
}
	Ошибка 300: [Семантическая] Функция main должна возвращать числовое значение, строка 3

Заключение

В ходе выполнения курсовой работы был разработан транслятор для языка программирования PAA-2022. Таким образом, были выполнены основные задачи данной курсовой работы:

- Сформулирована спецификация языка PAA-2022;
- Разработаны конечные автоматы и алгоритмы для реализация лексического анализатора;
- Разработана контекстно-свободная, приведённая к нормальной форме Грейбах, грамматика для описания синтаксически верных конструкций языка;
- Разработан семантический анализатор, осуществляющий проверку смысла используемых инструкций;
- Разработан транслятор с языка программирования PAA-2022 на язык низкого уровня Assembler;
- Проведено тестирование всех вышеперечисленных компонентов.

Окончательная версия языка PAA-2022 включает:

1) 2 типа данных;
2) Поддержка операции вывода;
3) 2 библиотечные функции
4) Возможность вызова функций стандартной библиотеки;
5) Наличие 5 арифметических операторов для вычисления выражений;
6) Структурированная система для обработки ошибок пользователя.
7) Условный оператор;
8) 6 операторов сравнения для целочисленного типа.

Список использованных источников

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

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

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

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

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

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

Приложение А

```
uint function factorial(uint x)
{
	new uint ret = 1;
	if (x < 0)
	{
		ret =  0;
	}
	if (x == 0)
	{
		ret = 1;
	}
	if (x > 0)
	{
		new uint y = x - 1;
		ret = factorial(y) * x;
	}
	return ret;
}
new uint function strCmp(string a, string b);
new uint function strLen(string a);
string function strCmpTest()
{
	new uint res = strCmp('aaa', 'bbb');
	new string outStr;
	if (res == 1)
	{
		outStr = 'Строка 1 больше строки 2';
	}
	if (res == 2)
	{
		outStr = 'Строка 2 больше строки 1';
	}
	if (res == 3)
	{
		outStr = 'Длина строк не совпадает, или передана пустая
строка';
	}
	return outStr;
}
main
{
	new uint x = strLen('hello');
	new uint y = factorial(x);
	output y;
	new string z = strCmpTest();
	output z;
	return 0;
}
```

Листинг 1 - Исходный код на языке РАА-2022

```
tfi<0>(ti<1>){dti<2>=l<3>;
e(i<1><l<4>){i<2>=l<4>;
}e(i<1>~l<4>){i<2>=l<3>;
}e(i<1>>l<4>){dti<5>=i<1>-l<3>;
i<2>=i<0>(i<5>)*i<1>;
}ri<2>;
}dtfi<6>(ti<7>,ti<8>);
dtfi<9>(ti<10>);
tfi<11>(){dti<12>=i<6>(l<13>,l<14>);
dti<15>;
e(i<12>~l<3>){i<15>=l<16>;
}e(i<12>~l<17>){i<15>=l<18>;
}e(i<12>~l<19>){i<15>=l<20>;
}ri<15>;
}m{dti<22>=i<9>(l<23>);
dti<24>=i<0>(i<22>);
pi<24>;
dti<25>=i<11>();
pi<25>;
rl<4>;
```

Листинг 2 - Таблица лексем на выходе лексического анализатора

------ Таблица идентификаторов ------

index	name	type	id type	scope	lexTable index
0	factorial	int	f	global	2
1	x	int	p	factorial	5
2	ret	int	v	factorial	10
3	L0	int	l	factorial	12
4	L1	int	l	factorial	18
5	y	int	v	scope_2	47
6	strCmp	int	f	global	70
7	a	str	p	strCmp	73
8	b	str	p	strCmp	76
9	strLen	int	f	global	82
10	a	str	p	strLen	85
11	strCmpTest	str	f	global	90
12	res	int	v	strCmpTest	96
13	L2	str	l	strCmpTest	100
14	L3	str	l	strCmpTest	102
15	outStr	str	v	strCmpTest	107
16	L4	str	l	scope_3	118
17	L5	int	l	strCmpTest	125
18	L6	str	l	scope_4	130
19	L7	int	l	strCmpTest	137
20	L8	str	l	scope_5	142
21	main	int	f	global	150
22	x	int	v	main	153
23	L9	str	l	main	157
24	y	int	v	main	162
25	z	str	v	main	174

Рисунок 3 - Таблица идентификаторов на выходе лексического анализатора

```
#define REG_DECLARE "new"
#define REG_STRING "string"
#define REG_INTEGER "uint"
#define REG_BOOL "bool"
#define REG_FUNCTION "function"
#define REG_RETURN "return"
#define REG_PRINT "output"
#define REG_MAIN "main"
#define REG_ID "([a-z]|[A-Z])+"
#define REG_INTEGER_LIT "([1-9]+[0-9]*)|0o?|([1-7]+[0-7]*o)"
#define REG_STRING_LIT "'(.)*'"
#define REG_BOOL_LIT "(true)|(false)"
#define REG_SEMICOLON ";"
#define REG_EQUAL "="
#define REG_COMMA ","
#define REG_MREQUAL "\\{"
#define REG_LSEQUAL "\\}"
#define REG_LEFTTHESIS "\\("
#define REG_RIGHTTHESIS "\\)"
#define REG_PLUS "\\+"
#define REG_MINUS "\\-"
#define REG_STAR "\\*"
#define REG_DIRSLASH "/"
#define REG_REM_AFTER_DIVIDING "%"
#define REG_IF "if"
#define REG_ELSE "else"
#define REG_MORE ">"
#define REG_LESS "<"
#define REG_EXCLAMATION "!"
```

Листинг 4 - Регулярные выражения для лексического распознавателя

Приложение Б

```
Greibach greibach(NS('S'), TS('$'),
    12,

    Rule(NS('S'), GRB_ERROR_SERIES + 0, 3,                      // Неверная структура программы
        Rule::Chain(6, TS('t'), TS('f'), TS('i'), NS('P'), NS('T'), NS('S')),
        Rule::Chain(2, TS('m'), NS('T')),
        Rule::Chain(7, TS('d'), TS('t'), TS('f'), TS('i'), NS('P'), TS(';'), NS('S'))
    ),

    Rule(NS('T'), GRB_ERROR_SERIES + 2, 2,                  // Ошибка в теле функции
        Rule::Chain(5, TS('{'), TS('r'), NS('V'), TS(';'), TS('}')),
        Rule::Chain(6, TS('{'), NS('K'), TS('r'), NS('V'), TS(';'), TS('}'))
    ),

    Rule(NS('P'), GRB_ERROR_SERIES + 1, 2,                  // Не найден список параметров функции
        Rule::Chain(3, TS('('), NS('E'), TS(')')),
        Rule::Chain(2, TS('('), TS(')'))
    ),

    Rule(NS('E'), GRB_ERROR_SERIES + 4, 2,                  // Ошибка в списке параметров функции
        Rule::Chain(4, TS('t'), TS('i'), TS(','), NS('E')),
        Rule::Chain(2, TS('t'), TS('i'))
    ),

    Rule(NS('F'), GRB_ERROR_SERIES + 5, 2,                  // Ошибка в вызове функции
        Rule::Chain(3, TS('('), NS('N'), TS(')')),
        Rule::Chain(2, TS('('), TS(')'))
    ),

    Rule(NS('N'), GRB_ERROR_SERIES + 6, 4,                  // Ошибка в списке параметров функции
        Rule::Chain(1, TS('i')),
        Rule::Chain(1, TS('l')),
        Rule::Chain(3, TS('i'), TS(','), NS('N')),
        Rule::Chain(3, TS('l'), TS(','), NS('N'))
    ),

    Rule(NS('Z'), GRB_ERROR_SERIES + 7, 5,                  // Ошибка в условии условного выражения
        Rule::Chain(5, TS('('), TS('i'), NS('L'), TS('i'), TS(')')),
        Rule::Chain(5, TS('('), TS('i'), NS('L'), TS('l'), TS(')')),
        Rule::Chain(5, TS('('), TS('l'), NS('L'), TS('i'), TS(')')),
        Rule::Chain(5, TS('('), TS('l'), NS('L'), TS('l'), TS(')')),
        Rule::Chain(3, TS('('), NS('V'), TS(')'))
    ),

    Rule(NS('L'), GRB_ERROR_SERIES + 8, 6,                  // Неверный условный оператор
        Rule::Chain(1, TS('<')),
        Rule::Chain(1, TS('>')),
        Rule::Chain(1, TS('$')),
        Rule::Chain(1, TS('#')),
        Rule::Chain(1, TS('~')),
        Rule::Chain(1, TS('&'))
    ),
```

Рисунок 1 – Грамматика языка PAA-2022 (продолжение)

```
Rule(NS('A'), GRB_ERROR_SERIES + 9, 5,                  // Неверный арифметический оператор
    Rule::Chain(1, TS('+')),
    Rule::Chain(1, TS('-')),
    Rule::Chain(1, TS('*')),
    Rule::Chain(1, TS('/')),
    Rule::Chain(1, TS('%'))
),

Rule(NS('V'), GRB_ERROR_SERIES + 10, 2,                 // Неверное выражение. Ожидаются только идентификаторы и литералы
    Rule::Chain(1, TS('l')),
    Rule::Chain(1, TS('i'))
),

Rule(NS('W'), GRB_ERROR_SERIES + 11, 10,                    // Ошибка в арифметичском выражении
    Rule::Chain(1, TS('i')),
    Rule::Chain(1, TS('l')),
    Rule::Chain(3, TS('('), NS('W'), TS(')')),
    Rule::Chain(5, TS('('), NS('W'), TS(')'), NS('A'), NS('W')),
    Rule::Chain(2, TS('i'), NS('F')),
    Rule::Chain(3, TS('i'), NS('A'), NS('W')),
    Rule::Chain(4, TS('i'), NS('F'), NS('A'), NS('W')),
    Rule::Chain(3, TS('l'), NS('A'), NS('W')),
    Rule::Chain(3, TS('i'), NS('L'), NS('W')),
    Rule::Chain(3, TS('l'), NS('L'), NS('W'))
),

Rule(NS('K'), GRB_ERROR_SERIES + 12, 16,                // Недопустимая синтаксическая конструкция
    Rule::Chain(7, TS('d'), TS('t'), TS('i'), TS('='), NS('W'), TS(';'), NS('K')),
    Rule::Chain(4, TS('i'), TS('='), NS('W'), TS(';')),
    Rule::Chain(5, TS('d'), TS('t'), TS('i'), TS(';'), NS('K')),
    Rule::Chain(5, TS('i'), TS('='), NS('W'), TS(';'), NS('K')),
    Rule::Chain(4, TS('p'), NS('V'), TS(';'), NS('K')),
    Rule::Chain(3, TS('{'), NS('K'), TS('}')),
    Rule::Chain(4, TS('{'), NS('K'), TS('}'), NS('K')),
    Rule::Chain(6, TS('e'), NS('Z'), TS('{') ,NS('K'), TS('}'), NS('K')),
    Rule::Chain(10, TS('e'), NS('Z'), TS('{') ,NS('K'), TS('}'), TS('y'), TS('{'),NS('K'), TS('}'), NS('K')),
    Rule::Chain(4, TS('i'), NS('F'), TS(';'), NS('K')),
    Rule::Chain(6, TS('d'), TS('t'), TS('i'), TS('='), NS('W'), TS(';')),   /
    Rule::Chain(4, TS('d'), TS('t'), TS('i'), TS(';')),
    Rule::Chain(3, TS('p'), NS('V'), TS(';')),
    Rule::Chain(5, TS('e'), NS('Z'), TS('{') ,NS('K'), TS('}')),
    Rule::Chain(9, TS('e'), NS('Z'), TS('{') ,NS('K'), TS('}'), TS('y'), TS('{'),NS('K'), TS('}')),
    Rule::Chain(3, TS('i'), NS('F'), TS(';'))
)
```

Рисунок 2 – Грамматика языка PAA-2022 (продолжение)

Приложение В

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

};
class my_stack_MfstState :public std::stack<MfstState> {
public:
    using std::stack<MfstState>::c;
};
struct Mfst                         //магазинный автомат
{
    enum RC_STEP {                  //код вовзрата функции step
        NS_OK,                  //найдено правило и цепочка, цепочка записана в стек
        NS_NORULE,              //не найдено правило грамматики(ошибка в грамматике)
        NS_NORULECHAIN,         //не найдена подходящая цепочка правила(ошибка в исходном коде)
        NS_ERROR,               //неизвестный нетерминальный символ грамматики
        TS_OK,                  //тек. символ ленты == вершине стека, продвинулась лента, pop стека
        TS_NOK,                 //тек. символ ленты != вершине стека, продвинулась лента, pop стека
        LENTA_END,              //текущая позиция ленты >= lenta_size
        SURPRISE,               //неожиданный код возврата (ошибка в step)
    };
```

Рисунок 1 – Структура магазинного автомата

```
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(short plenta_position, RC_STEP prc_step, short pnrule, short pnrule_chain);

}diagnosis[MFST_DIAGN_NUMBER];      // последние самые глубокие сообщения

GRBALPHABET* lenta;                 //перекодированныя (TN/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
LT::LexTable lex;                   //результат работы лексического анализатора
MFSTSTSTACK st;                     //стек автомата
my_stack_MfstState storestate;      //стек для хранения состояний
Mfst();
Mfst(LT::LexTable& plex, GRB::Greibach pgrebach, const std::string filePath);
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-ю строку диагностики или 0x00
bool savestate();                   //сохранить состояние автомата
bool resetstate();                  //восстановить состояние автомата
bool push_chain(GRB::Rule::Chain chain);        //поместить цепочку правила в стек
RC_STEP step();                     //выполнить шаг автомата
void start(std::ostream& outputStream);                     //запустить автомат
bool savediagnosis(RC_STEP pprc_step);          //код завершения шага
void printrules();                  //вывести послдеовательность правил
```

Рисунок 2 – Структура магазинного автомата (продолжение)

```
struct Deducation
{
    short size;
    short* nrules;
    short* nrulechains;
    Deducation()
    {
        size = 0;
        nrules = 0;
        nrulechains = 0;
    };
}deducation;

bool savededucation();
```

Рисунок 3 – Структура магазинного автомата (продолжение)

Приложение Г

```
Шаг : Правило            Входная лента                  Стек
0   : S->tfiPTS                                         S$
0   : SAVESTATE:         1
0   :                                                   tfiPTS$
1   :                    fi(ti){dti=l;e(i<l){i=l;}      fiPTS$
2   :                    i(ti){dti=l;e(i<l){i=l;}e      iPTS$
3   :                    (ti){dti=l;e(i<l){i=l;}e(      PTS$
4   : P->(E)             (ti){dti=l;e(i<l){i=l;}e(      PTS$
4   : SAVESTATE:         2
4   :                    (ti){dti=l;e(i<l){i=l;}e(      (E)TS$
5   :                    ti){dti=l;e(i<l){i=l;}e(i      E)TS$
6   : E->ti,E            ti){dti=l;e(i<l){i=l;}e(i      E)TS$
6   : SAVESTATE:         3
6   :                    ti){dti=l;e(i<l){i=l;}e(i      ti,E)TS$
7   :                    i){dti=l;e(i<l){i=l;}e(i~      i,E)TS$
8   :                    ){dti=l;e(i<l){i=l;}e(i~l      ,E)TS$
9   : 2
9   : RESSTATE
9   :                    ti){dti=l;e(i<l){i=l;}e(i      E)TS$
10  : E->ti              ti){dti=l;e(i<l){i=l;}e(i      E)TS$
10  : SAVESTATE:         3
10  :                    ti){dti=l;e(i<l){i=l;}e(i      ti)TS$
11  :                    i){dti=l;e(i<l){i=l;}e(i~      i)TS$
12  :                    ){dti=l;e(i<l){i=l;}e(i~l      )TS$
13  :                    {dti=l;e(i<l){i=l;}e(i~l)      TS$
14  : T->{rV;}           {dti=l;e(i<l){i=l;}e(i~l)      TS$
14  : SAVESTATE:         4
14  :                    {dti=l;e(i<l){i=l;}e(i~l)      {rV;}S$
15  :                    dti=l;e(i<l){i=l;}e(i~l){      rV;}S$
16  : 2
16  : RESSTATE
16  :                    {dti=l;e(i<l){i=l;}e(i~l)      TS$
17  : T->{KrV;}          {dti=l;e(i<l){i=l;}e(i~l)      TS$
17  : SAVESTATE:         4
17  :                    {dti=l;e(i<l){i=l;}e(i~l)      {KrV;}S$
18  :                    dti=l;e(i<l){i=l;}e(i~l){      KrV;}S$
19  : K->dti=W;K         dti=l;e(i<l){i=l;}e(i~l){      KrV;}S$
19  : SAVESTATE:         5
19  :                    dti=l;e(i<l){i=l;}e(i~l){      dti=W;KrV;}S$
20  :                    ti=l;e(i<l){i=l;}e(i~l){i      ti=W;KrV;}S$
21  :                    i=l;e(i<l){i=l;}e(i~l){i=      i=W;KrV;}S$
22  :                    =l;e(i<l){i=l;}e(i~l){i=l      =W;KrV;}S$
23  :                    l;e(i<l){i=l;}e(i~l){i=l;      W;KrV;}S$
24  : W->l               l;e(i<l){i=l;}e(i~l){i=l;      W;KrV;}S$
```

Листинг 1 – Начало разбора синтаксического анализатора

```
989 : W->iF               i();pi;rl;}}}}}}}}}}}}}}}     W;KrV;}$
989 : SAVESTATE:          76
989 :                     i();pi;rl;}}}}}}}}}}}}}}}     iF;KrV;}$
990 :                     ();pi;rl;}}}}}}}}}}}}}}}     F;KrV;}$
991 : F->(N)              ();pi;rl;}}}}}}}}}}}}}}}     F;KrV;}$
991 : SAVESTATE:          77
991 :                     ();pi;rl;}}}}}}}}}}}}}}}     (N);KrV;}$
992 :                     );pi;rl;}}}}}}}}}}}}}}}     N);KrV;}$
993 : TNS_NORULECHAIN/NS_NORULE
993 : RESSTATE
993 :                     ();pi;rl;}}}}}}}}}}}}}}}     F;KrV;}$
994 : F->()               ();pi;rl;}}}}}}}}}}}}}}}     F;KrV;}$
994 : SAVESTATE:          77
994 :                     ();pi;rl;}}}}}}}}}}}}}}}     ();KrV;}$
995 :                     );pi;rl;}}}}}}}}}}}}}}}     );KrV;}$
996 :                     ;pi;rl;}}}}}}}}}}}}}}}     ;KrV;}$
997 :                     pi;rl;}}}}}}}}}}}}}}}     KrV;}$
998 : K->pV;K             pi;rl;}}}}}}}}}}}}}}}     KrV;}$
998 : SAVESTATE:          78
998 :                     pi;rl;}}}}}}}}}}}}}}}     pV;KrV;}$
999 :                     i;rl;}}}}}}}}}}}}}}}     V;KrV;}$
1000: V->i                i;rl;}}}}}}}}}}}}}}}     V;KrV;}$
1000: SAVESTATE:          79
1000:                     i;rl;}}}}}}}}}}}}}}}     i;KrV;}$
1001:                     ;rl;}}}}}}}}}}}}}}}     ;KrV;}$
1002:                     rl;}}}}}}}}}}}}}}}     KrV;}$
1003: TNS_NORULECHAIN/NS_NORULE
1003: RESSTATE
1003:                     i;rl;}}}}}}}}}}}}}}}     V;KrV;}$
1004: TNS_NORULECHAIN/NS_NORULE
1004: RESSTATE
1004:                     pi;rl;}}}}}}}}}}}}}}}     KrV;}$
1005: K->pV;              pi;rl;}}}}}}}}}}}}}}}     KrV;}$
1005: SAVESTATE:          78
1005:                     pi;rl;}}}}}}}}}}}}}}}     pV;rV;}$
1006:                     i;rl;}}}}}}}}}}}}}}}     V;rV;}$
1007: V->i                i;rl;}}}}}}}}}}}}}}}     V;rV;}$
1007: SAVESTATE:          79
1007:                     i;rl;}}}}}}}}}}}}}}}     i;rV;}$
1008:                     ;rl;}}}}}}}}}}}}}}}     ;rV;}$
1009:                     rl;}}}}}}}}}}}}}}}     rV;}$
1010:                     l;}}}}}}}}}}}}}}}     V;}$
1011: V->l                l;}}}}}}}}}}}}}}}     V;}$
1011: SAVESTATE:          80
1011:                     l;}}}}}}}}}}}}}}}     l;}$
1012:                     ;}}}}}}}}}}}}}}}     ;}$
1013:                     }}}}}}}}}}}}}}}     }$
1014:                     }}}}}}}}}}}}}}}     $
1015: 6
1016: ------>LENTA_END
```

Листинг 2 – Конец разбора синтаксического анализатора

```
0   : S->tfiPTS
3   : P->(E)
4   : E->ti
7   : T->{KrV;}
8   : K->dti=W;K
12  : W->l
14  : K->eZ{K}K
15  : Z->(iLl)
17  : L-><
21  : K->i=W;
23  : W->l
26  : K->eZ{K}K
27  : Z->(iLl)
29  : L->~
33  : K->i=W;
35  : W->l
38  : K->eZ{K}
39  : Z->(iLl)
41  : L->>
45  : K->dti=W;K
49  : W->iAW
50  : A->-
51  : W->l
53  : K->i=W;
55  : W->iFAW
56  : F->(N)
57  : N->i
59  : A->*
60  : W->i
64  : V->i
67  : S->dtfiP;S
71  : P->(E)
72  : E->ti,E
75  : E->ti
79  : S->dtfiP;S
83  : P->(E)
84  : E->ti
88  : S->tfiPTS
91  : P->()
93  : T->{KrV;}
94  : K->dti=W;K
98  : W->iF
99  : F->(N)
100 : N->l,N
102 : N->l
105 : K->dti;K
109 : K->eZ{K}K
110 : Z->(iLl)
112 : L->~
116 : K->i=W;
118 : W->l
121 : K->eZ{K}K
```

```
122 : Z->(iLl)
124 : L->~
128 : K->i=W;
130 : W->l
133 : K->eZ{K}
134 : Z->(iLl)
136 : L->~
140 : K->i=W;
142 : W->l
146 : V->i
149 : S->mT
150 : T->{KrV;}
151 : K->dti=W;K
155 : W->iF
156 : F->(N)
157 : N->l
160 : K->dti=W;K
164 : W->iF
165 : F->(N)
166 : N->i
169 : K->pV;K
170 : V->i
172 : K->dti=W;K
176 : W->iF
177 : F->()
180 : K->pV;
181 : V->i
184 : V->l
```

Листинг 3 – Пример разбора синтаксического анализатора

Приложение Д

```
int Preorities(char operation) {
    if (operation == LEX_LEFTTHESIS || operation == LEX_RIGHTTHESIS) {
        return 1;
    }
    if (operation == LEX_FULL_EQUALS || operation == LEX_NOT_FULL_EQUALS) {
        return 2;
    }
    if (operation == LEX_LESS || operation == LEX_MORE ||
        operation == LEX_LESS_OR_EQUALS || operation == LEX_MORE_OR_EQUALS) {
        return 3;
    }
    if (operation == LEX_MINUS || operation == LEX_PLUS) {
        return 4;
    }
    if (operation == LEX_STAR || operation == LEX_DIRSLASH || LEX_REM_AFTER_DIVIDING) {
        return 5;
    }

}

void PN::ConvertToPolishNotation(LT::LexTable &lexTable, IT::IdTable &idtable, int index, LT::Entry* expression)
{
    std::stack<LT::Entry> stack;

    int expressionSize = 0;
    int startPosition = index;
    short leftThesis = 0;

    for (; lexTable.table[index].lexema != LEX_SEMICOLON; index++) {
        LT::Entry lex = lexTable.table[index];
        if ((lex.lexema == LEX_ID || lex.lexema == LEX_LITERAL) && idtable.table[lex.idxTI].idtype != IT::IDTYPE::F) {
            expression[expressionSize++] = lex;
        }
        else if (lex.lexema == LEX_LEFTTHESIS) {
            stack.push(lex);
```

Рисунок 1 - Реализация польской нотации

```
    }
    else if (lex.lexema == LEX_ID && idtable.table[lexTable.table[index].idxTI].idtype == IT::IDTYPE::F)
    {
        LT::Entry copy = lexTable.table[index];
        copy.lexema = LEX_COMMERCIAL_AT;
        copy.idxTI = lexTable.table[index].idxTI;
        ++index;
        int countOfParams = 0;
        while (lexTable.table[index].lexema != LEX_RIGHTTHESIS) {
            if (((lexTable.table[index].lexema != LEX_COMMA) && (lexTable.table[index].lexema == LEX_ID || lexTable.table[index].lexema == LEX_LITERAL)))
            {
                ++countOfParams;
                expression[expressionSize++] = lexTable.table[index];
            }
            ++index;
        }
        expression[expressionSize++] = copy;
        expression[expressionSize].lexema = std::to_string(countOfParams).front();
        expression[expressionSize++].idxTI = -1;
    }
    else if (stack.empty() || stack.top().lexema == LEX_LEFTTHESIS) {
        stack.push(lex);
    }
    else
    {
        while (stack.size()) {
            if (Preorities(lexTable.table[index].lexema) > Preorities(stack.top().lexema)) break;
            expression[expressionSize++] = stack.top();
            stack.pop();
        }
        stack.push(lexTable.table[index]);
    }
}
```

Рисунок 2 - Реализация польской нотации (продолжение)

```
    while (stack.size() != 0)
    {
        expression[expressionSize++] = stack.top();
        stack.pop();
    }
    for (int i = 0, j = startPosition; i < expressionSize; ++i, ++j) {
        lexTable.table[j] = expression[i];
    }
    for (int i = 0; i < index - (startPosition + expressionSize); ++i) {
        for (int j = startPosition + expressionSize; j < lexTable.size; ++j) {
            lexTable.table[j] = lexTable.table[j + 1];
        }
        --lexTable.size;
    }
    for (int i = 0; i < idtable.size; ++i) {
        if (idtable.table[i].idxfirstLE > startPosition)
            idtable.table[i].idxfirstLE -= index - (startPosition + expressionSize);
    }
}


void PN::PolishNotation(LT::LexTable &lexTable, IT::IdTable &idTable)
{
    for (auto i = 0; i < lexTable.size; ++i) {
        if (lexTable.table[i].lexema == LEX_EQUAL) {
            LT::Entry expression[100];
            PN::ConvertToPolishNotation(lexTable, idTable, i + 1, expression);
        }
    }
}
```

Рисунок 3 - Реализация польской нотации (продолжение)

```
tfi<0>(ti<1>){dti<2>=l<3>;
e(i<1><l<4>){i<2>=l<4>;
}e(i<1>~l<4>){i<2>=l<3>;
}e(i<1>>l<4>){dti<5>=i<1>l<3>-;
i<2>=i<5>@<0>1i<1>*;
}ri<2>;
}dtfi<6>(ti<7>,ti<8>);
dtfi<9>(ti<10>);
tfi<11>(){dti<12>=l<13>l<14>@<6>2;
dti<15>;
e(i<12>~l<3>){i<15>=l<16>;
}e(i<12>~l<17>){i<15>=l<18>;
}e(i<12>~l<19>){i<15>=l<20>;
}ri<15>;
}m{dti<22>=l<23>@<9>1;
dti<24>=i<22>@<0>1;
pi<24>;
dti<25>=@<11>0;
pi<25>;
rl<4>;
}
```

Листинг 4 - Таблица лексем после преобразования к польской нотации

Приложение Е

Сгенерированный код

```
.586
.model flat, stdcall
includelib libucrt.lib
includelib kernel32.lib
includelib ../Debug/PAA_Lib.lib
ExitProcess PROTO : DWORD
_strCmp PROTO : DWORD, :DWORD
_strLen PROTO : DWORD
_outStr PROTO : DWORD
_outBool PROTO : DWORD
_outInt PROTO : DWORD


.stack 4096
.const
	_DIVISION_BY_ZERO_ERROR BYTE 'Ошибка выполнения: деление на ноль',
0
	_OVERFLOW_ERROR BYTE 'Ошибка выполнения: переполнение', 0
	_NEGATIVE_RESULT_ERROR BYTE 'Ошибка выполнения: попытка присвоения
отрицательного значения', 0
	L0 DWORD 1
	L1 DWORD 0
	L2 BYTE 'aaa', 0
	L3 BYTE 'bbb', 0
	L4 BYTE 'Строка 1 больше строки 2', 0
	L5 DWORD 2
	L6 BYTE 'Строка 2 больше строки 1', 0
	L7 DWORD 3
	L8 BYTE 'Длина строк не совпадает, или передана пустая строка', 0
	L9 BYTE 'hello', 0
.data
	_factorialret		DWORD 0 ; uint
	_scope_2y		DWORD 0 ; uint
	_strCmpTestres			DWORD 0 ; uint
	_strCmpTestoutStr		DWORD ? ; str
	_mainx		DWORD 0 ; uint
	_mainy		DWORD 0 ; uint
	_mainz		DWORD ? ; str

.code
_factorial PROC _factorialx: DWORD
	push		L0
	pop			_factorialret

	cmp			_factorialret, 0
	jl			NEGATIVE_RESULT
	push		_factorialx
	push		L1
```

```
        pop             ebx
        pop             eax
        cmp             eax, ebx
        jnb             FALSE4
        push            L1
        pop                     _factorialret

        cmp                     _factorialret, 0
        jl                      NEGATIVE_RESULT
FALSE4:
        push            _factorialx
        push            L1
        pop             ebx
        pop             eax
        cmp             eax, ebx
        jne             FALSE8
        push            L0
        pop                     _factorialret

        cmp                     _factorialret, 0
        jl                      NEGATIVE_RESULT
FALSE8:
        push            _factorialx
        push            L1
        pop             ebx
        pop             eax
        cmp             eax, ebx
        jna             FALSE12
        push            _factorialx
        push            L0

        pop             ebx
        pop             eax
        sub             eax, ebx
        jo              EXIT_OVERFLOW
        push            eax

        pop                     _scope_2y

        cmp                     _scope_2y, 0
        jl                      NEGATIVE_RESULT
        push            _scope_2y
        call            _factorial
        push            eax
        push            _factorialx

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

```
	pop			_factorialret

	cmp			_factorialret, 0
	jl			NEGATIVE_RESULT
FALSE12:


	jmp EXIT
EXIT_DIV_ON_NULL:
	push offset _DIVISION_BY_ZERO_ERROR
	call _outStr
	push -1
	call ExitProcess

EXIT_OVERFLOW:
	push offset _OVERFLOW_ERROR
	call _outStr
	push -2
	call ExitProcess

NEGATIVE_RESULT:
	push offset _NEGATIVE_RESULT_ERROR
	call _outStr
	push -3
	call ExitProcess

EXIT:
	mov		eax, _factorialret
	ret		4
_factorial ENDP

_strCmpTest PROC
	push		offset L2
	push		offset L3
	call		_strCmp
	push		eax
	pop			_strCmpTestres

	cmp			_strCmpTestres, 0
	jl			NEGATIVE_RESULT
	push		_strCmpTestres
	push		L0
	pop		ebx
	pop		eax
	cmp		eax, ebx
	jne		FALSE25
	push		offset L4
	pop			_strCmpTestoutStr

FALSE25:
	push		_strCmpTestres
	push		L5
```

```
	pop		ebx
	pop		eax
	cmp		eax, ebx
	jne		FALSE29
	push		offset L6
	pop			_strCmpTestoutStr

FALSE29:
	push		_strCmpTestres
	push		L7
	pop		ebx
	pop		eax
	cmp		eax, ebx
	jne		FALSE33
	push		offset L8
	pop			_strCmpTestoutStr

FALSE33:


	jmp EXIT
EXIT_DIV_ON_NULL:
	push offset _DIVISION_BY_ZERO_ERROR
	call _outStr
	push -1
	call ExitProcess

EXIT_OVERFLOW:
	push offset _OVERFLOW_ERROR
	call _outStr
	push -2
	call ExitProcess

NEGATIVE_RESULT:
	push offset _NEGATIVE_RESULT_ERROR
	call _outStr
	push -3
	call ExitProcess

EXIT:
	mov		eax, _strCmpTestoutStr
	ret		0
_strCmpTest ENDP

main PROC
	push		offset L9
	call		_strLen
	push		eax
	pop			_mainx

	cmp			_mainx, 0
	jl			NEGATIVE_RESULT
```

```
	push		_mainx
	call		_factorial
	push		eax
	pop			_mainy

	cmp			_mainy, 0
	jl			NEGATIVE_RESULT
	push		_mainy
	call		_outInt

	call		_strCmpTest
	push		eax
	pop			_mainz

	push		_mainz
	call		_outStr



	jmp EXIT
EXIT_DIV_ON_NULL:
	push offset _DIVISION_BY_ZERO_ERROR
	call _outStr
	push -1
	call ExitProcess

EXIT_OVERFLOW:
	push offset _OVERFLOW_ERROR
	call _outStr
	push -2
	call ExitProcess

NEGATIVE_RESULT:
	push offset _NEGATIVE_RESULT_ERROR
	call _outStr
	push -3
	call ExitProcess

EXIT:
	push		L1
	call		ExitProcess

main ENDP
end main
```

Результат работы

```
120
Строка 2 больше строки 1
```