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 SIA-2020»

Выполнил студент Сачишин Иван Александрович
(Ф.И.О.)

Руководитель проекта преп.-стаж. Пахолко Алена Степановна
(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 преп.-стаж. Пахолко Алена Степановна
(учен. степень, звание, должность, подпись, Ф.И.О.)

Нормоконтролер преп.-стаж. Пахолко Алена Степановна
(учен. степень, звание, должность, подпись, Ф.И.О.)

Курсовой проект защищен с оценкой

Минск 2020

Содержание

Введение

Главной целью данной курсовой работы является разработка транслятора для языка программирования SIA-2020. Основная задача транслятора заключается в том, чтобы сделать программу, написанную языке программирования SIA-2020, понятной компьютеру. В данном курсовом проекте трансляция будет осуществляться в код на языке Assembler.

Транслятор SIA-2020 состоит из следующих частей:

– лексический анализатор;
– синтаксический анализатор;
– семантический анализатор;
– генератор исходного кода на языке ассемблера

Исходя из цели курсового проекта, были определены следующие задачи:

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

Язык программирования SIA-2020 предназначен для выполнения простейших арифметический действий и операций над строками.

1 Спецификация языка программирования

1.1 Характеристика языка программирования

Язык SIA-2020 – это универсальный, строго типизированный, процедурный, компилируемый язык. Не является объектно-ориентированным.

1.2 Алфавит языка

Алфавит языка SIA-2020 основан на кодировке Windows-1251, представленной на рисунке 1.1.

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

Рисунок 1.1 – Алфавит входных символов

Исходный код SIA-2020 может содержать символы латинского алфавита, цифры десятичной системы счисления от 0 до 9, русские символы разрешены только в строковых литералах.

1.3 Символы 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
{ }	Ограничивают программные конструкции условного оператора
;	Признак конца инструкции языка
[]	Ограничивает программные блоки
‘’	Ограничивают литерал

1.4 Применяемые кодировки

Для написания исходного кода на языке программирования SIA-2020 используется кодировка Windows-1251.

1.5 Типы данных

В языке SIA-2020 есть 2 типа данных: целочисленный и строковый. Описание типов данных, предусмотренных в данным языке представлено в таблице 1.2.

Таблица 1.2 – Типы данных языка SIA-2020

Тип данных	Описание типа данных
num	Фундаментальный тип данных. Предусмотрен для объявления целочисленных данных (4 байта). Автоматически инициализируется нулевым значением. Возможные операции: *Арифметические* + – бинарный, суммирование; - – бинарный, вычитание; * – бинарный, умножение; / – бинарный, деление; % – бинарный, остаток от деления; = – присваивание значения; & – бинарный, сравнение; ! – бинарный, сравнение;

Окончание таблицы 1.2

Тип данных	Описание типа данных
Word	Фундаментальный тип данных. Предусмотрен для объявления строк. (1 символ – 1 байт). К строкам, переменным и литералам операции не применяются. Автоматическая инициализация строкой нулевой длины. Максимальное количество символов в строке – 255.

1.6 Преобразование типов днных

В языке программирования SIA-2020 преобразование типов данных не поддерживается.

1.7 Идентификаторы

В имени идентификатора допускаются символы латинского алфавита нижнего регистра. Максимальная длина имени - 9 символов.

1.8 Литералы

В языке существует 2 типа литералов: целого и символьного типов. Краткое описание литералов представлено в таблице 1.3.

Таблица 1.3 – Описание литералов

Тип литерала	Регулярное выражение	Описание	Пример
Целочисленный литерал	[1-9]+[0-9]*	Целочисленные неотрицательные литералы, по умолчанию инициализируются 0. Литералы могут быть только rvalue.	create num ch; ch = 15; 15 – целочисленный литерал.
Строковые литерал	[a-z|A-Z|A-Я|a-я|0-9|!-/]+	Символы, заключённые в ‘’ (одинарные кавычки), по умолчанию инициализируются пустой строкой. Литералы могут быть только rvalue.	create word str; str = ‘Hello wold!’ ‘Hello world!’ – строковый литерал.

В языке SIA-2020 над литералами можно производить различные операции.

1.9 Область видимости идентификаторов

Область видимости «сверху вниз» (по принципу C++). В языке SIA-2020 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

Таблица 1.4 – Способы инициализации переменных

Вид инициализации	Примечание
create <тип данных> <идентификатор>;	Автоматическая инициализация: переменные типа num инициализируются нулём, переменные типа word – пустой строкой.
<идентификатор> = <значение>;	Присваивание переменной значения.

1.11 Инструкции языка

Все возможные инструкции языка программирования SIA-2020 представлены в общем виде в таблице 1.5.

Таблица 1.5 – Инструкции языка программирования SIA-2020

Инструкция	Запись на языке SIA-2020
Объявление переменной	create <тип данных> <идентификатор>;
Присваивание	<идентификатор> = <значение> | <идентификатор>;
Объявление внешней функции	<тип данных> proc <идентификатор> (<тип данных> <идентификатор>, …) […]
Точка входа	entry […];
Возврат значения из подпрограммы	out <идентификатор> | <литерал>;
Вывод данных	outStream (<идентификатор> | <литерал>);

Условный оператор	if (<имя переменной, литерал><условный оператор><имя переменной, литерал>)
	{
	…
	}
	else
	{
	…
	};
	Блок else не обязателен.

1.12 Операции языка

Язык программирования SIA-2020 может выполнять арифметические операции, представленные в таблице 1.6.

Таблица 1.6 – Приоритетности операций языка программирования SIA-2020

Операция	Приоритетность операции
( )	1 или 5
,	2
+ -	3
* / %	4

Максимальным значением приоритетности является “5”, минимальным “1” соответственно.

1.13 Выражения и их вычисления

Выражение языка программирования SIA-2020 – это совокупность переменных, литералов, вызовов функций, знаков операций, скобок, которая может быть вычислена в соответствии с синтаксисом языка.

1.14 Программные конструкции языка

Ключевые программные конструкции языка программирования SIA-2020 представлены чуть ниже в таблице 1.7.

Таблица 1.7 – Программные конструкции языка SIA-2020

Конструкция	Описание
Главная функция (точка входа в приложение)	entry [… out <имя переменной/литерал>;];
Функция	<тип> proc <идентификатор>(<тип> <идентификатор>) [… out <выражение>;];

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

1.15 Область видимости

В языке SIA-2020 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 SIA-2020, приведена в таблице 1.8.

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

Часть семантических проверок выполняется на этапе лексического анализа.

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
lexStrCmp	num	word x – строка word y – строка	Функция лексикографически сравнивает строку x со строкой y
stringLen	num	word x - строка	Функция вычисляет длину строки x
outStreamN	0	num x - число	Функция выводит на консоль число x
outStreamW	0	word x - строка	Функция выводит на консоль строку x

1.19 Ввод и вывод данных

Ввод данных не поддерживается языком программирования SIA-2020.

outStream (<идентификатор или литерал>); – вывод в стандартный поток вывода.

В зависимости от типа параметра определяется функция: outStreamW или outStreamN, которые входят в состав стандартной библиотеки и описаны в таблице 1.9.

1.20 Точка входа

Точкой входа является функция entry.

1.21 Препроцессор

Препроцессор в языке программирования SIA-2020 не предусмотрен.

1.22 Соглашения о вызовах

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

– все параметры функции передаются через стек;
– память освобождает вызываемый код;
– занесение в стек параметров идёт справа налево.

1.23 Объектный код

Язык программирования SIA-2020 транслируется в язык ассемблера.

1.24 Классификация сообщений транслятора

В случае возникновения ошибки в коде программы на языке SIA-2020 и выявления её транслятором в текущий файл протокола выводится сообщение. Их классификация сообщений приведена в таблице 1.10.

Таблица 1.10 – Классификация сообщений транслятора

Интервал	Описание ошибок
0-99	Системные ошибки
100-109	Ошибки параметров
110-115	Ошибки открытия и чтения файлов
116-129	Ошибки лексического анализа
600-699	Ошибки синтаксического анализа
700-799	Ошибки семантического анализа

В SIA-2020 есть возможность модификации таблицы ошибок.

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 SIA-2020 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.

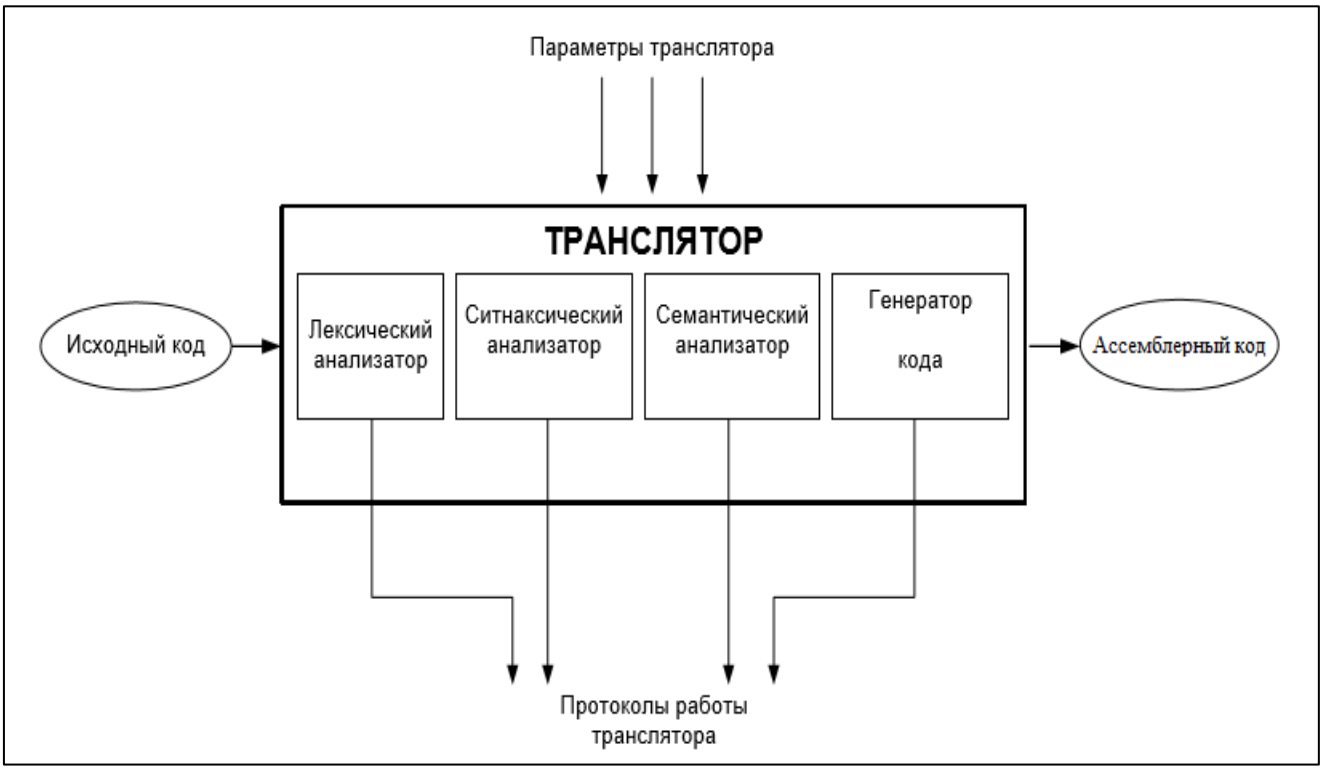

Рисунок 2.1 – Структура транслятора

Лексический анализ – первая фаза трансляции. Назначением лексического анализатора является нахождение ошибок лексики языка и формирование таблицы лексем и таблицы идентификаторов. Подробнее описан в 3 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. Подробное описание представлено в 5 главе.

В моём трансляторе 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 SIA-2020, прошедший все предыдущие этапы, в код на языке Ассемблера. Более полно описан в главе 7.

2.2 Перечень входных параметров транслятора

Входные параметры представлены в таблице 2.1.

Таблица 2.1 – Входные параметры транслятора языка SIA-2020

Входной параметр	Описание	Значение по умолчанию
-in:<имя_файла>	Входной файл с расширением .txt, в котором содержится исходный код на SIA-2020	Не предусмотрено
-log:<имя_файла>	Файл для записи результата проверки входного файла на допустимость символов	<имя_файла>.log
-sin:<имя_файла>	Файл для записи результата синтаксического разбора	<имя_файла>.sin

Обязательным является только параметр -in, остальные, при отсутсвии соответствующего ключа, будут формироваться из имени файла с исходным кодом и дефолтного расширения.

2.3 Перечень протоколов, формируемых транслятором и их содержимое

Таблица с перечнем протоколов, формируемых транслятором языка SIA-2020 и их назначением представлена в таблице 2.2

Таблица 2.2 – Протоколы, формируемые транслятором языка SIA-2020

Формируемый протокол	Описание протокола
Файл журнала с параметром <log>	Содержит информацию о входных параметрах в приложение и о этапе проверки символов на допустимость.
SIA_ASM.asm	Содержит сгенерированный код на языке Ассемблера.
<имя_файла>.lex.txt	Результат работы лексического анализатора. Содержит таблицы лексем.
<имя_файла>.id.txt	Результат работы лексического анализатора. Содержит таблицы идентификаторов.
<имя_файла>.sin	Результат работы синтаксического анализа. Содержит правила разбора, а так же трассировку.

При отсутствии соответствующих ключей файлы протоколов создаются автоматически.

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 SIA-2020. На выходе формируется таблица лексем и таблица идентификаторов. Структура лексического анализатора представлена на рисунке 3.1

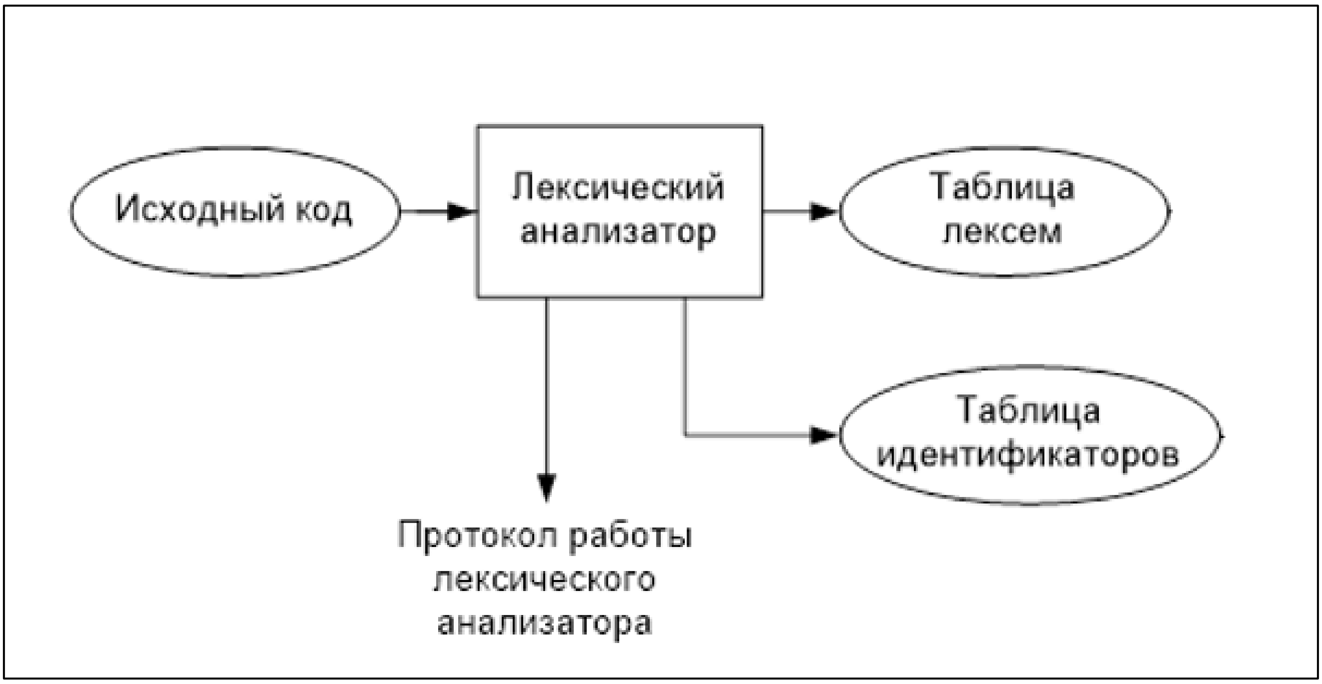

Рисунок 3.1 – Структура лексического анализатора SIA-2020

3.2 Контроль входных символов

Таблица для контроля входных символов представлена на рисунке 3.2

```
#define IN_CODE_TABLE {\
    /*0     1      2      3      4      5      6      7      8      9      A      B      C      D      E      F*/\
    IN::E, IN::F, IN::F, IN::F, IN::F, IN::F, IN::F, IN::F, IN::F, IN::S, 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::S, IN::V, IN::F, IN::F, IN::F, IN::V, IN::V, IN::Q, IN::B, IN::B, IN::V, IN::V, IN::B, IN::V, IN::F, IN::V,\
    IN::T, IN::T, IN::T, IN::T, IN::T, IN::T, IN::T, IN::T, IN::T, IN::T, IN::F, IN::B, IN::V, IN::V, IN::V, IN::F,\
    IN::F, IN::F, IN::F, IN::T, IN::F, IN::F, IN::F, IN::F, IN::F, IN::F, IN::F, IN::F, IN::T, IN::F, IN::F, IN::F,\
    IN::F, IN::F, IN::F, IN::T, IN::F, IN::F, IN::F, IN::F, IN::F, IN::F, IN::F, IN::B, IN::F, IN::B, 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::B, IN::F, IN::B, IN::F, IN::F,\
    \
    IN::F, IN::F, IN::F, IN::F, IN::F, IN::F, IN::F, IN::F, IN::F, IN::F, IN::F, IN::F, IN::F, IN::F, IN::F, IN::F,\
    IN::F, IN::F, IN::F, IN::F, IN::F, IN::F, IN::F, IN::F, IN::F, IN::F, IN::F, IN::F, IN::F, IN::F, IN::F, IN::F,\
    IN::F, IN::F, IN::F, IN::F, IN::F, IN::F, IN::F, IN::F, IN::F, IN::F, IN::F, IN::F, IN::F, IN::F, IN::F, IN::F,\
    IN::F, IN::F, IN::F, IN::F, IN::F, IN::F, IN::F, IN::F, IN::F, IN::F, IN::F, IN::F, IN::F, IN::F, IN::F, IN::F,\
    IN::F, IN::F, IN::F, IN::F, IN::F, IN::F, IN::F, IN::F, IN::F, IN::F, IN::F, IN::F, IN::F, IN::F, IN::F, IN::F,\
    IN::F, IN::F, IN::F, IN::F, IN::F, IN::F, IN::F, IN::F, IN::F, IN::F, IN::F, IN::F, IN::F, IN::F, IN::F, IN::F,\
    IN::F, IN::F, IN::F, IN::F, IN::F, IN::F, IN::F, IN::F, IN::F, IN::F, IN::F, IN::F, IN::F, IN::F, IN::F, IN::F,\
    IN::F, IN::F, IN::F, IN::F, 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

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, S – сепаратор, V – арифметический символ, Q – символ ограничивающий литерал, B – блокообразующие символы, E – символ конца файла.

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, сепараторов, символов операций и соответствующих им лексем, регулярных выражений и конечных автоматов

Лексемы – это символы, соответствующие ключевым словам, символам операций и сепараторам, необходимые для упрощения дальнейшей обработки исходного кода программы. Данное соответствие описано в таблице 3.1.

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

Тип цепочки	Цепочка	Лексема
Ключевые слова	create	c
	num	n
	word	w
	proc	p
	outStream	r
	lexStrCmp	x
	stringLen	s
	out	o
	entry	e
	if	m
	else	a
Иное	Идентификатор	i
	Целочисленный литерал	l

Окончание таблицы 3.1

Тип цепочки	Цепочка	Лексема
Сепараторы	;	;
	,	,
	[	[
	]	]
	(	(
	)	)
	=	=
	{	{
	}	}
Операторы	-	v
	+	v
	*	v
	/	v
	%	v
	&	&
	!	!

Каждому выражению соответствует детерминированный конечный автомат, то есть автомат с конечным состоянием, по которому происходит разбор данного выражения. На каждый автомат в массиве подаётся фраза и с помощью регулярного выражения, соответствующего данному графу переходов, происходит разбор. В случае успешного разбора выражения оно записывается в таблицу лексем. Если выражение является идентификатором или литералом, информация также заносится в таблицу идентификаторов. Пример реализации таблицы лексем представлен в приложении А. Пример реализации таблицы лексем представлен в приложении А.

Также в приложении А находятся конечные автоматы, соответствующие лексемам языка SIA-2020.

3.5 Основные структуры данных

Основные структуры таблиц лексем и идентификаторов данных языка SIA-2020, используемых для хранения, представлены в приложении А. В таблице лексем содержится лексема, её номер, полученный при разборе и номер строки в исходном коде. В таблице идентификаторов содержится имя идентификатора, номер в таблице лексем, тип данных, смысловой тип идентификатора и его значение. На первом этапе лексического анализа исходный текст программы разбивается на отдельные токены, которые хранятся в объекте класса words.

Класс words приведен на рисунке 3.3.

```
struct Word
{
    char token[257];
    int line;
    int symInLine;
};

class Words
{
public:
    list<Word> arr;
    void Add(Word entry)
    {
        arr.push_back(entry);
    }
    void Delete()
    {
        arr.pop_front();
    }
};
```

Рисунок 3.3 – Код класса хранения токенов

3.6 Принцип обработки ошибок

При возникновении критической ошибки – работа транслятора прекращается.

3.7 Структура и перечень сообщений лексического анализатора

Перечень сообщений лексического анализатора представлен на рисунке 3.4.

```
ERROR_ENTRY(116, "[LEX_ANALYSIS] Кол-во открывающих и закрывающих процедурных скобок не совпадает"),
ERROR_ENTRY(117, "[LEX_ANALYSIS] Таблица лексем переполнена"),
ERROR_ENTRY(118, "[LEX_ANALYSIS] Таблица идентификаторов переполнена"),
ERROR_ENTRY(119, "[LEX_ANALYSIS] Не удалось создать файл с лексемами / идентификаторами"),
ERROR_ENTRY(120, "[LEX_ANALYSIS] Превышена длинна стркового литерала"),
ERROR_ENTRY(121, "[LEX_ANALYSIS] Превышена длинна идентификатора"),
ERROR_ENTRY(122, "[LEX_ANALYSIS] Запрещенные символы в имени идетификатора"),
ERROR_ENTRY(123, "[LEX_ANALYSIS] Слишком большое значение целочисленного литерала"),
ERROR_ENTRY(124, "[LEX_ANALYSIS] Не объявлена точка входа в программу ( entry )"),
ERROR_ENTRY(125, "[LEX_ANALYSIS] Определено несколько точек входа в программу ( entry )"),
ERROR_ENTRY(126, "[LEX_ANALYSIS] Использование необъявленного идентификатора"),
ERROR_ENTRY(127, "[LEX_ANALYSIS] Неверное оъявление параметров"),
ERROR_ENTRY(128, "[LEX_ANALYSIS] Не найдена закрывающая кавычка строкового литерала"),
ERROR_ENTRY(129, "[LEX_ANALYSIS] Запрещенный элемент в глобальной области видимости"),
ERROR_ENTRY(130, "[LEX_ANALYSIS] Повторное определение идентификатора"),
```

Рисунок 3.4 – Перечень ошибок лексического анализатора

Если лексический анализ выполнен удачно, то на консоль выведется сообщение Lexical analysis done.

3.8 Параметры лексического анализатора и режимы его работы

Входным параметром лексического анализа является объект класса words, содержащий в себе динамическую структуру данных, которая содержит в себе исходный текст разбитый на токены. Каждый элемент в этом объекте представляет собой структуру, полями которых являются лексема, номер её строки в исходном файле и позицию в строке, полученные на этапе разбиения исходного текста на отдельные слова.

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: ‘num’.

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

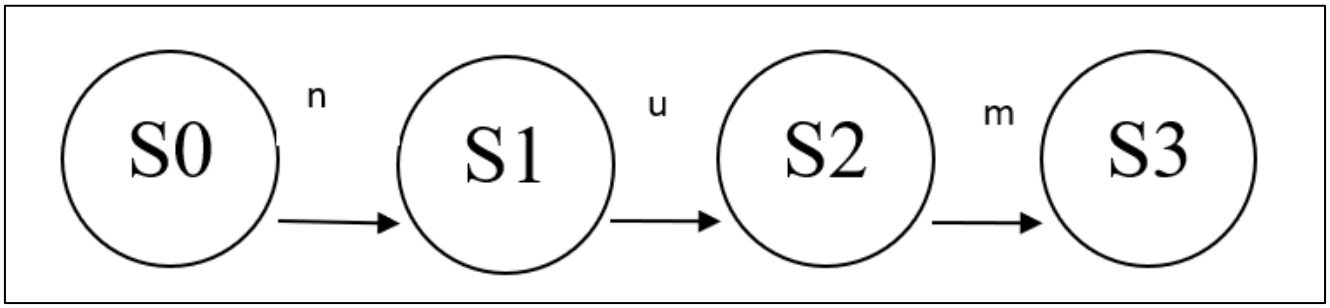

Рисунок 3.5 – Граф переходов для цепочки ‘num’

3.10 Контрольный пример

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

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.

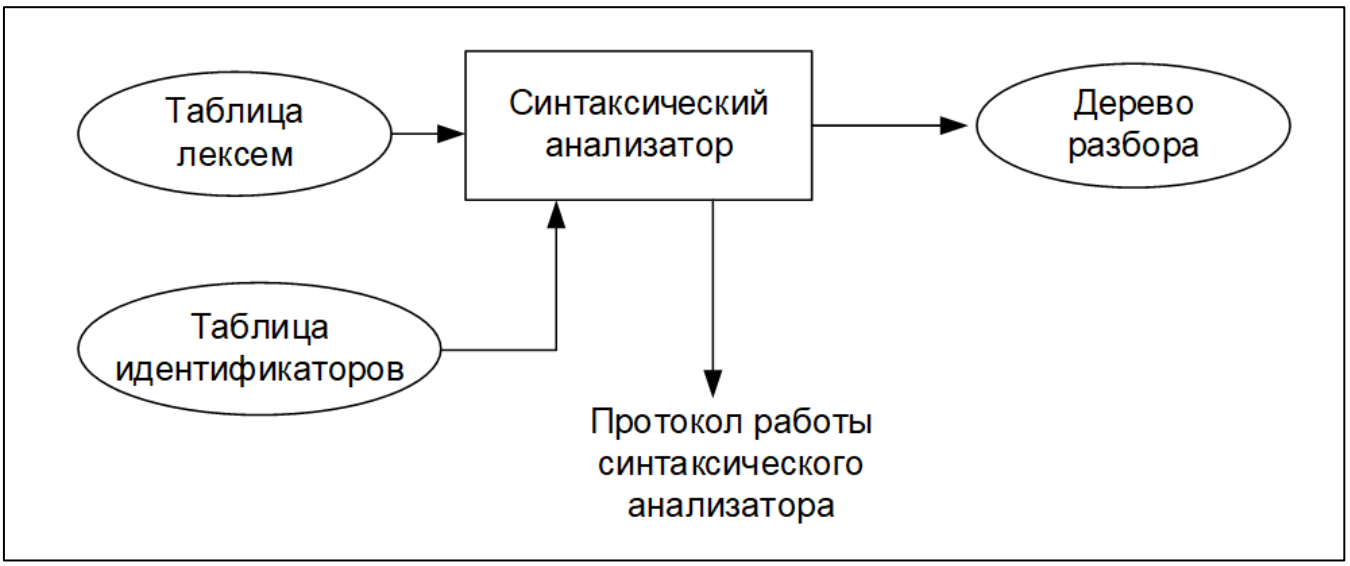

Рисунок 4.1 – Структура синтаксического анализатора

4.2 Контекстно свободная грамматика, описывающая синтаксис языка

В синтаксическом анализаторе транслятора языка SIA-2020 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 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.

Грамматика языка SIA-2020 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 SIA-2020 (назначни в таблицу)

Нетерминал	Цепочки правил	Описание
S	e[NoE;]; tpi(F)[N];S tpi(F)[N];	Порождает правила, описывающее общую структуру программы

N	cti;N
	i=E;N
	rE;N
	oE;N
	i(W);N
	s(W);N
	x(W);N
	mQ{N}a{N};N
	mQ{N};N
	cti;
	rE;
	s(W);
	i=E;
	oE;
	x(W);
	i(W);
	mQ{N};
	mQ{N}a{N};
		Порождает правила, описывающие конструкции языка

E	i l iM lM i(W) (E) (E)M i(W)M s(W)M x(W)M s(W) x(W)	Порождает правила, описывающие выражения
F	ti ti, F	Описывают параметры функции при объявлении
W	i l i, W l, W	Порождает правила, описывающие принимаемые параметры функции

Нетерминал	Цепочки правил	Описание
M	vE vEM	Порождает правила, описывающие знаки арифметических операций
Q	(E<E) (E>E) (E&E) (E!E)	Порождает правила условия в условных конструкциях.

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
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 и структуру грамматики Грейбах, описывающей правила языка SIA-2020. 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 рисунке 4.1.

```
ERROR_ENTRY(600, "[SYNTAX_ANALYSIS] Неверная структура программы"),
ERROR_ENTRY(601, "[SYNTAX_ANALYSIS] Ошибочный оператор"),
ERROR_ENTRY(602, "[SYNTAX_ANALYSIS] Ошибка в выражении"),
ERROR_ENTRY(603, "[SYNTAX_ANALYSIS] Ошибка в выражении со знаком функции"),
ERROR_ENTRY(604, "[SYNTAX_ANALYSIS] Ошибка в формальных параметрах функции"),
ERROR_ENTRY(605, "[SYNTAX_ANALYSIS] Ошибка в фактических параметрах функции"),
ERROR_ENTRY(606, "[SYNTAX_ANALYSIS] Ошибка в условии условного оператора вызываемой функции"),
```

Рисунок 4.1 – Перечень сообщений синтаксического анализатора

Если синтаксический анализ выполнен без ошибок, то на консоль выведется сообщение Syntax analysis done.

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 SIA-2020 представлен в приложении Г. Дерево разбора исходного кода также представлено в приложении Г

5 Разработка семантического анализатора

5.1 Структура семантического анализатора

Семантический анализ языка SIA-2020 выполняется после выполнения лексического и синтаксического анализа. Несмотря на это, некоторые семантические проверки выполняются на этапе лексического анализа. На вход семантического анализатора подаются таблица лексем и таблица идентификаторов.

5.2 Функции семантического анализатора

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

5.3 Структура и перечень сообщений семантического анализатора

Сообщения, формируемые семантическим анализатором, представлены на рисунке 5.1.

```
ERROR_ENTRY(700, "[SEMANTIC ANALYSIS] Идентификатор не является левосторонним выражением"),
ERROR_ENTRY(701, "[SEMANTIC ANALYSIS] Оператор не перегружен для работы со строками"),
ERROR_ENTRY(702, "[SEMANTIC ANALYSIS] В вызове функции отсутствуют скруглые скобки ()"),
ERROR_ENTRY(703, "[SEMANTIC ANALYSIS] Несоответствие типов в выражении"),
ERROR_ENTRY(704, "[SEMANTIC ANALYSIS] Ошибка в выражении"),
ERROR_ENTRY(705, "[SEMANTIC ANALYSIS] Превышено максимальное кол-во параметров функции"),
ERROR_ENTRY(706, "[SEMANTIC ANALYSIS] Превышено максимальное кол-во функции"),
ERROR_ENTRY(707, "[SEMANTIC ANALYSIS] Несоответствие формальных и фактических параметров функции"),
ERROR_ENTRY(708, "[SEMANTIC ANALYSIS] Несоответствие формальных и фактических параметров встроенной функции"),
ERROR_ENTRY(709, "[SEMANTIC ANALYSIS] Тип возвращаемого значения не соответствует типу функции"),
ERROR_ENTRY(710, "[SEMANTIC ANALYSIS] В функции отсутствует возвращаемое значение"),
ERROR_ENTRY(711, "[SEMANTIC ANALYSIS] Превышено максимальное значение int (4 byte)"),
ERROR_ENTRY(712, "[SEMANTIC ANALYSIS] Неверное выражение с outStream"),
ERROR_ENTRY(713, "[SEMANTIC ANALYSIS] Неверное выражение в блоке if"),
ERROR_ENTRY(714, "[SEMANTIC ANALYSIS] Главная функция entry может возвращать только значение типа INT"),
```

Рисунок 5.1 – Перечень сообщений семантического анализатора

Если семантический анализ выполнен без ошибок, то на консоль выведется сообщение Sematic analysis done.

5.4 Принцип обработки ошибок

При обнаружении ошибки в исходном коде программы семантический анализатор формирует сообщение об ошибки 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.

6 Преобразование выражений

6.1 Выражения, допускаемые языком

В языке SIA-2020 допускаются выражения, применимые к целочисленным типам данных. В выражениях поддерживаются арифметические операции, такие как +, -, *, /, % и (), и вызовы функций как операнды арифметических выражений.

Приоритет операций представлен в таблице 6.1.

Таблица 6.1 – Приоритет операций в языке SIA-2020

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

Выражения в языке SIA-2020 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 которого записывается информация о вызываемой функции, а в поле приоритета для данной лексемы записывается число параметров вызываемой функции;

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

Таблица 6.2 – Пример преобразования выражения в обратную польскую запись

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

Использование польской записи позволяет вычислить выражение за один проход.

6.3 Программная реализация обработки выражений

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

6.4 Контрольный пример

Пример использования польской записи представлен в таблице 6.2. Преобразование выражений в формат польской записи необходимо для построения более простых алгоритмов их вычисления.

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.

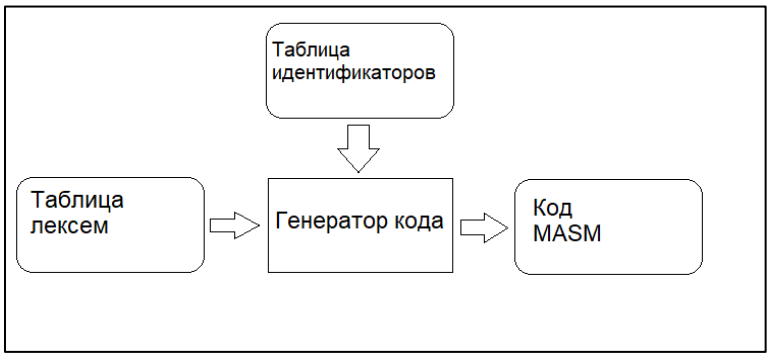

Рисунок 7.1 – Структура генератора кода

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

Обобщенная блок-схема алгоритма генерации кода языка ассемблера изображена на рисунке 7.2.

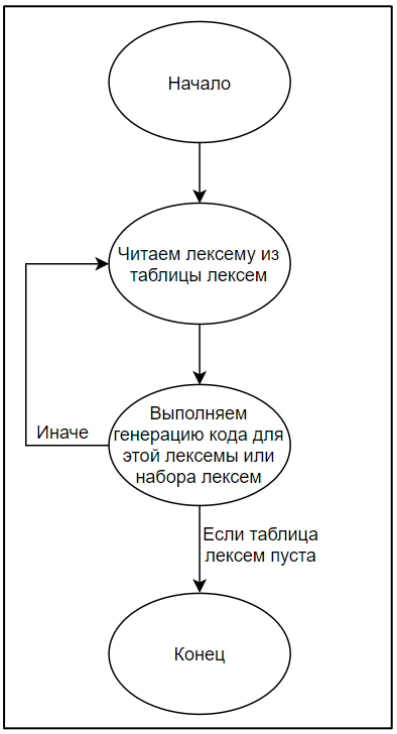

Рисунок 7.2 – Блок-схема алгоритма генерации кода языка ассемблер

7.2 Представление типов данных в оперативной памяти

Элементы таблицы идентификаторов расположены в разных сегментах языка ассемблера – .data и .const. Идентификаторы языка SIA-2020 размещены в сегменте данных(.data). Литералы – в сегменте констант (.const). Соответствия между типами данных идентификаторов на языке SIA-2020 и на языке ассемблера приведены в таблице 7.1.

Таблица 7.1 – Соответствия типов идентификаторов языка SIA-2020 и языка Ассемблера

Тип идентификатора на языке SIA-2020	Тип идентификатора на языке ассемблера	Пояснение
num	SDWORD	Хранит целочисленный тип данных со знаком.
word	DWORD	Хранит указатель на начало строки.
L(0-9)	BYTE SDWORD	Литералы: строковые, целочисленные

7.3 Алгоритм работы генератора кода

Преобразования происходят по принципу, встретив определённую лексему и зная, в каком месте программы находится сейчас лексема, программа генерирует код на языке Ассемблера.

Генерируемый код записывается в файл, который размещен в заранее подготовленном проекте. Сгенерированный код можно посмотреть в приложении Е.

7.4 Контрольный пример

Генерируемый код записывается в файл «SIA_Asm.asm». Сгенерированный код можно посмотреть в приложении Е.

8 Тестирование транслятора

8.1 Тестирование фазы проверки на допустимость символов

В языке SIA-2020 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
Num proc test ”num a”	Ошибка 111:[SYSTEM] Недопустимый символ в исходном коде (-in) Строка 1 | Позиция: 14

8.2 Тестирование лексического анализатора

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
entry [ы];	Ошибка 111:[SYSTEM] Недопустимый символ в исходном коде (-in) Строка 3 | Позиция: 1
entry [x = 5;];	Ошибка 126:[LEX_ANALYSIS] Использование необъявленного идентификатора Строка 2 | Позиция: 2
entry [num create hello;];	Ошибка 126:[LEX_ANALYSIS] Использование необъявленного идентификатора Строка 2 | Позиция: 6
hello [ ];	Ошибка 124:[LEX_ANALYSIS] Не объявлена точка входа в программу (entry)
entry [entry[];];	Ошибка 125:[LEX_ANALYSIS] Определено несколько точек входа в программу (entry) Строка 2 | Позиция: 1
entry [Create num nameoverflow;];	Ошибка 121:[LEX_ANALYSIS] Превышена длинна идентификатора Строка 2 | Позиция: 5

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

8.3 Тестирование синтаксического анализатора

На этапе синтаксического анализа могут возникнуть ошибки, описанные в пункте 4.6. Результаты тестирования синтаксического анализатора показаны в таблице 8.3.

Таблица 8.3 – Тестирование синтаксического анализатора

Исходный код	Диагностическое сообщение
num proc test (a, num b)	Ошибка 127:[LEX_ANALYSIS] Неверное оъявление параметров Строка 0 | Позиция: 6
num proc eq(num a) [out 2;];	Ошибка 600 : [SYNTAX_ANALYSIS] Неверная структура программы Строка 4 | Позиция 1 Ошибка 602 : [SYNTAX_ANALYSIS] Ошибка в выражении Строка 3 | Позиция 5 Ошибка 602 : [SYNTAX_ANALYSIS] Ошибка в выражении Строка 3 | Позиция 5

if(1 3)
{
…
}
	Ошибка 602 : [SYNTAX_ANALYSIS] Ошибка в выражении
	Строка 2 | Позиция 6
	Ошибка 602 : [SYNTAX_ANALYSIS] Ошибка в выражении
	Строка 2 | Позиция 6
	Ошибка 602 : [SYNTAX_ANALYSIS] Ошибка в выражении
	Строка 2 | Позиция 6

8.4 Тестирование семантического анализатора

Итоги тестирования семантического анализатора приведены в таблице 8.4.

Таблица 8.4 – Тестирование семантического анализатора

Исходный код	Диагностическое сообщение
entry [create word qa; qa = ‘123’ + 5;];	Ошибка 703:[SEMANTIC ANALYSIS] Несоответствие типов в выражении Строка 8 | Позиция: 0
entry [create word qa; qa = ‘123’ + 5; out ‘bb’;];	Ошибка 714:[SEMANTIC ANALYSIS] Главная функция entry может возвращать только значение типа INT Строка 5 | Позиция: 0

Заключение

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

- Сформулирована спецификация языка SIA-2020;
- 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 SIA-2020 на язык низкого уровня Assembler;
- Проведено тестирование всех вышеперечисленных компонентов.

Окончательная версия языка SIA-2020 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) 4 оператора 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.

7. ASCII Table and Description [Электронный ресурс] – Режим доступа: http://www.asciitable.com/ - Дата доступа: 02.10.2020

Приложение А

```
num proc factorial(num x)
[
	if(x<0)
	{
		out 0;
	};
	if(x&0)
	{
		out 1;
	}
	else
	{
		create num temp;
		temp = x - 1;
		out factorial(temp)*x;
	};
];
num proc sunlight(word a, word b)
[
	create num q;
	q = lexStrCmp(a,b);
	out q;
];
entry
[
	create word stroka;
	create num len;
	create num result;
	stroka = 'Hello wolrd!';
	len = stringLen(stroka);
	result = sunlight(stroka, 'Hello wolrd!');
	outStream result;
	outStream factorial(5)*2;
	outStream len;
	out 0;
];
```

Исходный код на языке SIA-2020

```
1	tpi(ti)
2	[
3	m(i<l)
4	{
5	ol;
6	};
7	m(i&l)
8	{
9	ol;
10	}
11	a
```

```
12      {
13      cti;
14      i=ivl;
15      oi(i)vi;
16      };
17      ];
18      tpi(ti,ti)
19      [
20      cti;
21      i=x(i,i);
22      oi;
23      ];
24      e
25      [
26      cti;
27      cti;
28      cti;
29      i=l;
30      i=s(i);
31      i=i(i,l);
32      ri;
33      ri(l)vl;
34      ri;
35      ol;
36      ];
```

Таблица лексем на выходе лексического анализатора

Литералы

Тип данных:	Значение:	Длина строки:
INT	0	-
INT	1	-
INT	1	-
INT	2	-
INT	0	-
STR	'Числа одинаковы'	17
STR	'Числа различны'	16
INT	15	-
INT	35	-
STR	'Нау'	5

Функции

Идентификатор:	Тип возвращаемого значения:
sum	INT
isequals	INT
getlen	INT
stringLen	INT

Параметры

Параметр:	Родительский блок:	Тип данных:
asum	sum	INT
bsum	sum	INT
aisequals	isequals	INT
bisequals	isequals	INT
stringgetlen	getlen	STR

Переменные

Имя родительского блока:	Идентификатор:	Тип данных:	Тип идентификатора:	Значение:	Длина строки:
ENTRY	rescmpENTRY	INT	V	0	-
ENTRY	ressumENTRY	INT	V	0	-

Таблица идентификаторов

```
#define FST_COUNT 11

#define A_PROC 5,          \
            FST::NODE(1, FST::RELATION('p', 1)),  \
            FST::NODE(1, FST::RELATION('r', 2)),  \
            FST::NODE(1, FST::RELATION('o', 3)),  \
            FST::NODE(1, FST::RELATION('c', 4)),  \
            FST::NODE()

#define A_NUM 4,\
            FST::NODE(1, FST::RELATION('n', 1)),\
            FST::NODE(1, FST::RELATION('u', 2)),\
            FST::NODE(1, FST::RELATION('m', 3)),\
            FST::NODE()

#define A_CREATE 7,\
            FST::NODE(1, FST::RELATION('c', 1)),\
            FST::NODE(1, FST::RELATION('r', 2)),\
            FST::NODE(1, FST::RELATION('e', 3)),\
            FST::NODE(1, FST::RELATION('a', 4)),\
            FST::NODE(1, FST::RELATION('t', 5)),\
            FST::NODE(1, FST::RELATION('e', 6)),\
            FST::NODE()

#define A_OUT 4,\
            FST::NODE(1, FST::RELATION('o', 1)),\
            FST::NODE(1, FST::RELATION('u', 2)),\
            FST::NODE(1, FST::RELATION('t', 3)),\
            FST::NODE()


#define A_WORD 5,                \
            FST::NODE(1, FST::RELATION('w', 1)),  \
            FST::NODE(1, FST::RELATION('o', 2)),  \
            FST::NODE(1, FST::RELATION('r', 3)),  \
            FST::NODE(1, FST::RELATION('d', 4)),  \
            FST::NODE()
```

```
#define A_OUTSTREAM 10,              \
            FST::NODE(1, FST::RELATION('o', 1)),  \
            FST::NODE(1, FST::RELATION('u', 2)),  \
            FST::NODE(1, FST::RELATION('t', 3)),  \
            FST::NODE(1, FST::RELATION('S', 4)),  \
            FST::NODE(1, FST::RELATION('t', 5)),  \
            FST::NODE(1, FST::RELATION('r', 6)),  \
            FST::NODE(1, FST::RELATION('e', 7)),  \
            FST::NODE(1, FST::RELATION('a', 8)),  \
            FST::NODE(1, FST::RELATION('m', 9)),  \
            FST::NODE()

#define A_ENTRY 6,                \
            FST::NODE(1, FST::RELATION('e', 1)),\
            FST::NODE(1, FST::RELATION('n', 2)),\
            FST::NODE(1, FST::RELATION('t', 3)),\
            FST::NODE(1, FST::RELATION('r', 4)),\
            FST::NODE(1, FST::RELATION('y', 5)),\
            FST::NODE()

#define A_IF 3,\
            FST::NODE(1, FST::RELATION('i', 1)),\
            FST::NODE(1, FST::RELATION('f', 2)),\
            FST::NODE()

#define A_ELSE 5,\
            FST::NODE(1, FST::RELATION('e', 1)),\
            FST::NODE(1, FST::RELATION('l', 2)),\
            FST::NODE(1, FST::RELATION('s', 3)),\
            FST::NODE(1, FST::RELATION('e', 4)),\
            FST::NODE()
```

```
#define A_STRCMP 10,\
            FST::NODE(1, FST::RELATION('l', 1)),\
            FST::NODE(1, FST::RELATION('e', 2)),\
            FST::NODE(1, FST::RELATION('x', 3)),\
            FST::NODE(1, FST::RELATION('S', 4)),\
            FST::NODE(1, FST::RELATION('t', 5)),\
            FST::NODE(1, FST::RELATION('r', 6)),\
            FST::NODE(1, FST::RELATION('C', 7)),\
            FST::NODE(1, FST::RELATION('m', 8)),\
            FST::NODE(1, FST::RELATION('p', 9)),\
            FST::NODE()

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

Структуры конечных автоматов для лексического распознавателя

```
struct Entry    //строка таблицы лексем
{
    char lexema;    //лексема
    int sn;         //номер строки в исходном тексте
    int idxTI;      //индекс в таблице идентификаторов или LT_TI_NULLIDX
    int posInStr;
};

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
    void writeLT(const wchar_t* in);
};
```

Структура строки и самой таблицы лексем

```
enum IDDATATYPE { DEF = 0, INT = 1, STR = 2 };  //т
enum IDTYPE { V = 1, F = 2, P = 3, L = 4 , A = 5};
struct Entry    //строка таблицы идентификаторов
{
    char parrentBlock[ID_MAXSIZE+1];    //!!!
    int         idxfirstLE;     //индекс первой стр
    char        id[2*ID_MAXSIZE];       //идентифик
    IDDATATYPE  iddatatype;     //тип данных
    IDTYPE      idtype;     //тип идентификатора
    struct VALUE
    {
        int vint = NULL;    //значение integer
        struct
        {
            int len;    //кол-во символов в string
            char str[TI_STR_MAXSIZE];   //символы s
        } vstr;     //значение string
    } value;
};

struct IdTable      //экземпляр таблицы идентификат
{
    int maxsize;    //емкость таблицы идентификатор
    int size;   //текущий размер таблицы идентифика
    Entry* table;   //массив строк таблицы идентифи
    void writeIT(const wchar_t* in);
};
```

Структура строки и самой таблицы идентификаторов

Приложение Б

```
Greibach greibach(NS('S'), TS('$')              // стартовый символ / дно стека
    , 7
    , Rule(NS('S'), GRB_ERROR_SERIES + 0, 3,        // структура программы
        Rule::Chain(11, TS('t'), TS('p'), TS('i'), TS('('), NS('F'), TS(')'), TS('['), NS('N'), TS(']'), TS(';'), NS('S')),
        Rule::Chain(8, TS('e'), TS('['), NS('N'), TS('o'), NS('E'), TS(';'), TS(']'), TS(';')),
        Rule::Chain(10, TS('t'), TS('p'), TS('i'), TS('('), NS('F'), TS(')'), TS('['), NS('N'), TS(']'), TS(';')))

    , Rule(NS('N'), GRB_ERROR_SERIES + 1, 18        // конструкции в функциях
        , Rule::Chain(5, TS('c'), TS('t'), TS('i'), TS(';'), NS('N'))
        , Rule::Chain(5, TS('i'), TS('='), NS('E'), TS(';'), NS('N'))
        , Rule::Chain(4, TS('r'), NS('E'), TS(';'), NS('N'))
        , Rule::Chain(4, TS('o'), NS('E'), TS(';'), NS('N'))
        , Rule::Chain(6, TS('i'), TS('('), NS('W'), TS(')'), TS(';'), NS('N'))
        , Rule::Chain(6, TS('s'), TS('('), NS('W'), TS(')'), TS(';'), NS('N'))
        , Rule::Chain(6, TS('x'), TS('('), NS('W'), TS(')'), TS(';'), NS('N'))
        , Rule::Chain(11,TS('m'), NS('Q'), TS('{'), NS('N'), TS('}'), TS('a'), TS('{'), NS('N'), TS('}'),  TS(';'), NS('N'))
        , Rule::Chain(7, TS('m'), NS('Q'), TS('{'), NS('N'), TS('}'),TS(';'), NS('N'))
        , Rule::Chain(4, TS('c'), TS('t'), TS('i'), TS(';'))
        , Rule::Chain(3, TS('r'), NS('E'), TS(';'))
        , Rule::Chain(5, TS('s'), TS('('), NS('W'), TS(')'), TS(';'))
        , Rule::Chain(4, TS('i'), TS('='), NS('E'), TS(';'))
        , Rule::Chain(3, TS('o'), NS('E'), TS(';'))
        , Rule::Chain(5, TS('x'), TS('('), NS('W'), TS(')'), TS(';'))
        , Rule::Chain(5, TS('i'), TS('('), NS('W'), TS(')'), TS(';'))
        , Rule::Chain(6, TS('m'), NS('Q'), TS('{'), NS('N'), TS('}'), TS(';'))
        , Rule::Chain(10, TS('m'), NS('Q'), TS('{'), NS('N'), TS('}'), TS('a'), TS('{'), NS('N'), TS('}'), TS(';')))
```

```
, Rule(NS('E'), GRB_ERROR_SERIES + 2, 12        // выражения
    , Rule::Chain(1, TS('i'))
    , Rule::Chain(1, TS('l'))
    , Rule::Chain(2, TS('i'), NS('M'))
    , Rule::Chain(2, TS('l'), NS('M'))
    , Rule::Chain(4, TS('i'), TS('('), NS('W'), TS(')'))
    , Rule::Chain(3, TS('('), NS('E'), TS(')'))
    , Rule::Chain(4, TS('('), NS('E'), TS(')'), NS('M'))
    , Rule::Chain(5, TS('i'), TS('('), NS('W'), TS(')'), NS('M'))
    , Rule::Chain(5, TS('s'), TS('('), NS('W'), TS(')'), NS('M'))
    , Rule::Chain(5, TS('x'), TS('('), NS('W'), TS(')'), NS('M'))
    , Rule::Chain(4, TS('s'), TS('('), NS('W'), TS(')'))
    , Rule::Chain(4, TS('x'), TS('('), NS('W'), TS(')')))


, Rule(NS('M'), GRB_ERROR_SERIES + 3, 2     // знаки
    , Rule::Chain(2, TS('v'), NS('E'))
    , Rule::Chain(3, TS('v'), NS('E'), NS('M')))

, Rule(NS('F'), GRB_ERROR_SERIES + 4, 2     // формальные параметры функции
    , Rule::Chain(2, TS('t'), TS('i'))
    , Rule::Chain(4, TS('t'), TS('i'), TS(','), NS('F')))

, Rule(NS('W'), GRB_ERROR_SERIES + 5, 4     //  фактические параметры функции
    , 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('Q'), GRB_ERROR_SERIES + 6, 4     //  Условие условного оператора
    , Rule::Chain(5, TS('('), NS('E'), TS('<'), NS('E'), TS(')'))
    , Rule::Chain(5, TS('('), NS('E'), TS('>'), NS('E'), TS(')'))
    , Rule::Chain(5, TS('('), NS('E'), TS('&'), NS('E'), TS(')'))
    , Rule::Chain(5, TS('('), NS('E'), TS('!'), NS('E'), TS(')')))
```

Экземпляр грамматики в нормальной форме Грейбах

Приложение В

```
struct MfstState
{
    short lenta_position;
    short nrule;
    short nrulechain;
    MFSTSTSTACK st;
    MfstState();
    MfstState(
        short pposition,
        MFSTSTSTACK pst,
        short pnrulechain);

    MfstState(
        short pposition,
        MFSTSTSTACK pst,
        short pnrule,
        short pnrulechain);
};
```

```
struct Mfst
{
    enum RC_STEP
    {
        NS_OK,
        NS_NORULE,
        NS_NORULECHAIN,
        NS_ERROR,
        TS_OK,
        TS_NOK,
        LENTA_END,
        SURPRICE
    };

    struct MfstDiagnosis
    {
        short lenta_position;
        RC_STEP rc_step;
        short nrule;
        short nrule_chain;
        MfstDiagnosis();
        MfstDiagnosis(
            short plenta_position,
            RC_STEP prc_step,
            short pnrule,
            short pnrule_chain);
    } diagnosis[MFST_DIAGN_NUMBER];
```

```
GRBALPHABET* lenta;                                 //перекодирован
int FST_TRACE_n = 0;
short lenta_position;                               //текущая позиц
short nrule;                                        //номер текущег
short nrulechain;                                   //номер текущей
short lenta_size;                                   //размер ленты
GRB::Greibach grebach;                              //грамматика Гр
Scanner::Tables Scanner;                                    //резул
MFSTSTSTACK st;                                     //стек автомата
std::stack<MfstState> storestate;                   //стек для сохр
Mfst();
Mfst(
    Scanner::Tables plex,                                  //результ
    GRB::Greibach pgrebach);                          //грамматика
char* getCSt(char* buf);                              //получить со
char* getCLenta(char* buf, short pos, short n = 25);//лента:
char* getDiagnosis(char* buf);                      //получить n-ую
bool savestate(Log::LOG log);
bool restate(Log::LOG log);
bool push_chain(                                  //поместить цепочку
    GRB::Rule::Chain chain);                      //цепочка правила
RC_STEP step(Log::LOG log);                                 //выполн
bool start(Log::LOG log);                                   //запуст
bool savediagnosis(
    RC_STEP pprc_step);                          //код завершения шага
void printrules(Log::LOG& log);                             //вывести

struct Deducation                                   //вывод
{
    short size;                                   //количество шагов
    short* nrules;                                //номера правил гра
    short* nrulechains;                           //номера цепочек пр
    Deducation() { size = 0; nrules = 0; nrulechains = 0; }
} deducation;
bool savededucation();                              //сохранить дере
```

```
Lex::LEX lex;                       // результат работы лексического анализатора
MFSTSTSTACK stack;                          // стек автомата
std::stack<MfstState> storestate;   // стек для сохранения состояний
Mfst();
Mfst(
    Lex::LEX plex,          // результат работы лексического анализатора
    GRB::Greibach pgrebach          // грамматика Грейбах
);
char* getContainStack(char* buf);           // получить содержимое стека
char* getCLenta(char* buf, short pos, short n = 25);    // лента: n символов с pos
char* getDiagnosis(short n, char* buf);     // получить n-ую строку диагностики или 0x00
bool saveState(Log::LOG log);                   // сохранить состояние автомата
bool restState(Log::LOG log);                   // восстановить состояние автомата
bool push_chain(                        // поместить цепочку правила в стек
    GRB::Rule::Chain chain      // цепочка правил
);
Mfst::RC_STEP step(Log::LOG log);// выполнить шаг автомата
bool start(Log::LOG log);                       // запустить автомат
bool saveDiagnosis(
    RC_STEP pprc_step   // код завершения шага
);
void printRules(Log::LOG& log); // вывести последовательность правил (дерево разбора)
```

Структуры автомата синтаксического анализатора

Приложение Г

Начало разбора

```
Шаг : Правило                Входная лента                    Стек
1   : S->tpi(F)[N];S         tpi(ti,ti)[oivi;];tpi(ti,        S$
1   : SAVESTATE:             1
1   :                        tpi(ti,ti)[oivi;];tpi(ti,        tpi(F)[N];S$
2   :                        pi(ti,ti)[oivi;];tpi(ti,t        pi(F)[N];S$
3   :                        i(ti,ti)[oivi;];tpi(ti,ti        i(F)[N];S$
4   :                        (ti,ti)[oivi;];tpi(ti,ti)        (F)[N];S$
5   :                        ti,ti)[oivi;];tpi(ti,ti)[        F)[N];S$
6   : F->ti                  ti,ti)[oivi;];tpi(ti,ti)[        F)[N];S$
6   : SAVESTATE:             2
6   :                        ti,ti)[oivi;];tpi(ti,ti)[        ti)[N];S$
7   :                        i,ti)[oivi;];tpi(ti,ti)[m        i)[N];S$
8   :                        ,ti)[oivi;];tpi(ti,ti)[m(        )[N];S$
9   : TS_NOK/NS_NORULECHAIN
9   : RESTATE
9   :                        ti,ti)[oivi;];tpi(ti,ti)[        F)[N];S$
10  : F->ti,F                ti,ti)[oivi;];tpi(ti,ti)[        F)[N];S$
10  : SAVESTATE:             2
10  :                        ti,ti)[oivi;];tpi(ti,ti)[        ti,F)[N];S$
11  :                        i,ti)[oivi;];tpi(ti,ti)[m        i,F)[N];S$
12  :                        ,ti)[oivi;];tpi(ti,ti)[m(        ,F)[N];S$
13  :                        ti)[oivi;];tpi(ti,ti)[m(i        F)[N];S$
14  : F->ti                  ti)[oivi;];tpi(ti,ti)[m(i        F)[N];S$
14  : SAVESTATE:             3
14  :                        ti)[oivi;];tpi(ti,ti)[m(i        ti)[N];S$
15  :                        i)[oivi;];tpi(ti,ti)[m(i&        i)[N];S$
16  :                        )[oivi;];tpi(ti,ti)[m(i&i        )[N];S$
17  :                        [oivi;];tpi(ti,ti)[m(i&i)        [N];S$
18  :                        oivi;];tpi(ti,ti)[m(i&i){        N];S$
19  : N->oE;N                oivi;];tpi(ti,ti)[m(i&i){        N];S$
19  : SAVESTATE:             4
19  :                        oivi;];tpi(ti,ti)[m(i&i){        oE;N];S$
20  :                        ivi;];tpi(ti,ti)[m(i&i){o        E;N];S$
21  : E->i                   ivi;];tpi(ti,ti)[m(i&i){o        E;N];S$
```

Конец разбора

```
751 :                          i(l);ol;];                          E;oE;];$
752 : E->iM                    i(l);ol;];                          E;oE;];$
752 : SAVESTATE:               47
752 :                          i(l);ol;];                          iM;oE;];$
753 :                          (l);ol;];                           M;oE;];$
754 : TNS_NORULECHAIN/NS_NORULE
754 : RESTATE
754 :                          i(l);ol;];                          E;oE;];$
755 : E->i(W)                  i(l);ol;];                          E;oE;];$
755 : SAVESTATE:               47
755 :                          i(l);ol;];                          i(W);oE;];$
756 :                          (l);ol;];                           (W);oE;];$
757 :                          l);ol;];                            W);oE;];$
758 : W->l                     l);ol;];                            W);oE;];$
758 : SAVESTATE:               48
758 :                          l);ol;];                            l);oE;];$
759 :                          );ol;];                             );oE;];$
760 :                          ;ol;];                              ;oE;];$
761 :                          ol;];                               oE;];$
762 :                          l;];                                E;];$
763 : E->l                     l;];                                E;];$
763 : SAVESTATE:               49
763 :                          l;];                                l;];$
764 :                          ;];                                 ;];$
765 :                          ];                                  ];$
766 :                          ;                                   ;$
767 :                                                              $
768 : LENTA_END
769 : ------->LENTA_END
 ----------------------------------------------------------------------
```

Пример трассировки синтаксического анализатора

Правила синтаксического разбора

```
0   : S->tpi(F)[N];S
4   : F->ti
8   : N->mQ{N};N
9   : Q->(E<E)
10  : E->i
12  : E->l
15  : N->oE;
16  : E->l
20  : N->mQ{N}a{N};
21  : Q->(E&E)
22  : E->i
24  : E->l
27  : N->oE;
28  : E->l
33  : N->cti;N
37  : N->i=E;N
39  : E->iM
40  : M->vE
41  : E->l
43  : N->oE;
44  : E->i(W)M
46  : W->i
48  : M->vE
49  : E->i
55  : S->tpi(F)[N];S
59  : F->ti,F
62  : F->ti
66  : N->cti;N
70  : N->i=E;N
72  : E->x(W)
74  : W->i,W
76  : W->i
79  : N->oE;
80  : E->i
84  : S->e[NoE;];
86  : N->cti;N
90  : N->cti;N
94  : N->cti;N
98  : N->i=E;N
100 : E->l
102 : N->i=E;N
104 : E->s(W)
106 : W->i
109 : N->i=E;N
111 : E->i(W)
113 : W->i,W
115 : W->l
118 : N->rE;N
119 : E->i
121 : N->rE;N
122 : E->i(W)M
```

```
124 : W->l
126 : M->vE
127 : E->l
129 : N->rE;
130 : E->i
133 : E->l
```

Приложение Д

Реализация польской нотации

```
void PolishNotation(Scanner::Tables& tables)
{
    Pos pos;
    for (int i = 0; i < tables.lexTable.size; i++)
    {
        if (SEARCHEXPR)
        {
            pos.startEx = ++i;
            GetExrp(tables.lexTable, pos);
            ShiftAndWriteLT(tables, &pos, Convertation(pos, &tables.idenTable));
        }
    }
    cout << "Polish notation is done" << endl;
}

void GetExrp(LT::LexTable lexTable, Pos& pos)
{
    LT::Entry* expr = nullptr;
    for (pos.endEx = pos.startEx; lexTable.table[pos.endEx].lexema != LEX_SEMICOLON; pos.endEx++);
    expr = new LT::Entry[pos.endEx - pos.startEx];
    for (int i = pos.startEx, j = 0; i < pos.endEx; i++, j++)
    {
        expr[j] = lexTable.table[i];
    }
    pos.express = expr;
}
```

```
int Convertation(Pos& pos, IT::IdTable* idenTable)
{
    stack<LT::Entry> stack;
    int len = pos.endEx - pos.startEx;
    LT::Entry* arr = new LT::Entry[len];
    int countComma = 0;
    int brBalance = 0;
    int j = 0;
    for (int i = 0; i < len; i++)
    {
        switch (pos.express[i].lexema)
        {
            case LEX_STRCMP:
            case LEX_STRLEN:
            case LEX_ID:
            case LEX_LITERAL:
            {
                arr[j++] = pos.express[i];
                break;
            }

            case LEX_ARITHMETIC:
            {
                if (stack.empty() || stack.top().lexema==LEX_LEFTHESIS)
                {
                    stack.push(pos.express[i]);
                    break;
                }

                if (ArithmPrioritys(pos.express[i], idenTable)<= ArithmPrioritys(stack.top(), idenTable))
                {
                    while(!stack.empty()&&ArithmPrioritys(pos.express[i], idenTable) <= ArithmPrioritys(stack.top(), idenTable))
                    {
                        arr[j++] = stack.top();
                        stack.pop();
                    }
                }
                stack.push(pos.express[i]);
```

```
                stack.push(pos.express[i]);

                break;
            }

            case LEX_COMMA:
            {
                countComma++;
                if (ArithmPrioritys(pos.express[i], idenTable) < ArithmPrioritys(stack.top(), idenTable))
                {
                    do
                    {
                        arr[j++] = stack.top();
                        stack.pop();
                    } while (ArithmPrioritys(pos.express[i], idenTable) < ArithmPrioritys(stack.top(), idenTable));
                }
                break;
            }

            case LEX_LEFTHESIS:
            {
                stack.push(pos.express[i]);
                break;
            }

            case LEX_RIGHTHESIS:
            {
                int k = i;
                for (; k != 0; k--)
                {
                    if (pos.express[k].lexema == LEX_LEFTHESIS)
                    {
                        brBalance--;
                        if (idenTable->table[pos.express[k - 1].idxTI].idtype == IT::F && brBalance==0)
                        {
                            arr[j].lexema = '@';
                            arr[j].sn = pos.express[k].sn;
                            arr[j++].idxTI = ++countComma;
```

```
                    arr[j++].idxTI = ++countComma;
                    countComma = 0;
                }
                break;
            }
            else if (pos.express[k].lexema == LEX_RIGHTHESIS)brBalance++;
        }
        brBalance = 0;
        while(stack.top().lexema!=LEX_LEFTHESIS)
        {
            arr[j++] = stack.top();
            stack.pop();
        }
        stack.pop();
        break;
    }
}
}
while (!stack.empty())
{
    arr[j++] = stack.top();
    stack.pop();
}
```

#define LEX_RIGHT
лексема для)
Расширяется в:')'
Поиск в Интернете

```
                stack.pop();
                break;
            }
        }
    }
    while (!stack.empty())
    {
        arr[j++] = stack.top();
        stack.pop();
    }
    LT::Entry* resStr = new LT::Entry[j];
    for (int i = 0; i < j; i++)
    {
        resStr[i] = arr[i];
    }
    delete[] arr;
    delete[] pos.express;
    pos.express = resStr;
    return j;
}

short ArithmPrioritys(LT::Entry str, IT::IdTable* idenTable)
{
    switch (str.lexema)
    {
        case LEX_LEFTHESIS:
        case LEX_RIGHTHESIS:
        {
            return 1;
```

```
        case LEX_COMMA:
        {
            return 2;
        }

        case LEX_ARITHMETIC:
        {
            if (idenTable->table[str.idxTI].id[0] == LEX_PLUS || idenTable->table[str.idxTI].id[0] == LEX_MINUS)
                return 3;
            else if (idenTable->table[str.idxTI].id[0] == LEX_STAR || idenTable->table[str.idxTI].id[0] == LEX_DIRSLASH)
                return 4;
        }


    }
    return 0;
}

void ShiftAndWriteLT(Scanner::Tables& tables, Pos* pos, int newLen)
{
    for (int i = pos->startEx, j = 0; j < newLen; i++, j++)
    {
        tables.lexTable.table[i] = pos->express[j];
        if (tables.lexTable.table[i].lexema == LEX_ID || tables.lexTable.table[i].lexema == LEX_LITERAL || tables.lexTable.table[i].lexema == LEX_STRCMP ||
         tables.lexTable.table[i].lexema == LEX_STRLEN)
        {
            tables.idenTable.table[tables.lexTable.table[i].idxTI].idxfirstLE = i;
        }
    }


    for (int i = pos->endEx - (pos->startEx + newLen), j = 0; j < i; j++)
    {
        for (int k = pos->startEx+newLen; k < tables.lexTable.size; k++)
        {
            tables.lexTable.table[k] = tables.lexTable.table[k + 1];
            if (tables.lexTable.table[k].lexema == LEX_ID || tables.lexTable.table[k].lexema == LEX_LITERAL || tables.lexTable.table[k].lexema == LEX_STRCMP ||
             tables.lexTable.table[k].lexema == LEX_STRLEN)
            {
                tables.idenTable.table[tables.lexTable.table[k].idxTI].idxfirstLE = k;
            }
        }
        tables.lexTable.size--;
    }
}
```

Таблица лексем и таблица идентификаторов после преобразования к польской нотации

```
1       tpi(ti)
2       [
3       m(i<l)
4       {
5       ol;
6       };
7       m(i&l)
8       {
9       ol;
10      }
11      a
12      {
13      cti;
14      i=ilv;
15      oii@iv;
16      };
17      ];
18      tpi(ti,ti)
19      [
20      cti;
21      i=xii@;
22      oi;
23      ];
24      e
```

```
25      [
26      cti;
27      cti;
28      cti;
29      i=l;
30      i=si@;
31      i=iil@;
32      ri;
33      ril@lv;
34      ri;
35      ol;
36      ];
```

№	Идентификатор	Тип данных	Тип идентификатора	Индекс в ТЛ	Значение
0	sum	real	функция	101	-
1	asum	real	параметр	12	-
2	bsum	real	параметр	13	-
3	+	real	Арифм. Оп.	13	-
4	isequals	real	функция	72	-
5	aisequals	real	параметр	23	-
6	bisequals	real	параметр	26	-
7	L0	real	литерал	37	0
8	L1	real	литерал	43	1
9	getlen	real	функция	110	-
10	stringgetlen	word	параметр	59	-
11	stringLen	real	функция	58	-
12	rescmpENTRY	real	переменная	79	0
13	L2	real	литерал	73	1
14	L3	real	литерал	74	2
15	L4	real	литерал	115	0
16	L5	word	литерал	85	[17]"'Числа одинаковы'"
17	L6	word	литерал	91	[16]"'Числа различны'"
18	ressumENTRY	real	переменная	107	0
19	L7	real	литерал	102	15
20	L8	real	литерал	103	35
21	L9	word	литерал	111	[5]"'Нау'"

Приложение E

Сгенерированный код

```
.586
.model flat, stdcall
includelib libucrt.lib
includelib kernel32.lib
includelib ..\Debug\LIB.lib
ExitProcess PROTO :DWORD
outStreamW PROTO: SDWORD
outStreamN PROTO: DWORD
stringLen PROTO: DWORD
lexStrCmp PROTO: DWORD, :DWORD
.stack 4096
.CONST
null_division BYTE 'ERROR: DIVISION BY ZERO', 0
       OVER_FLOW BYTE 'ERROR: OVERFLOW', 0
       L0 SDWORD 0
       L1 SDWORD 1
       L2 BYTE 'Hello wolrd!', 0
       L3 SDWORD 5
       L4 SDWORD 2
       L5 SDWORD 0
.DATA
       tempfactorial SDWORD 0
       qsunlight SDWORD 0
       strokaENTRY DWORD ?
       lenENTRY SDWORD 0
       resultENTRY SDWORD 0
.CODE
factorial PROC xfactorial:SDWORD
       mov eax, xfactorial
       cmp eax, L0
              jl ifi1
              jge else1
ifi1:
       push L0
       jmp local0
else1:
       mov eax, xfactorial
       cmp eax, L0
              jz ifi2
              jnz else2
ifi2:
       push L1
       jmp local0
```

```
        jmp ifEnd2
else2:
        push xfactorial
        push L1
        pop ebx
        pop eax
        sub eax, ebx
        push eax
        pop eax
        mov tempfactorial, eax
        push tempfactorial
        call factorial
        push eax
        push xfactorial
        pop eax
        pop ebx
        mul ebx
        push eax
        jmp local0
ifEnd2:
local0:
        pop eax
        ret
factorial ENDP
sunlight PROC bsunlight:DWORD, asunlight:DWORD
        push asunlight
        push bsunlight
        call lexStrCmp
        push eax
        pop eax
        mov qsunlight, eax
        push qsunlight
        jmp local1
local1:
        pop eax
        ret
sunlight ENDP
main PROC
        push offset L2
        pop strokaENTRY
        push strokaENTRY
        call stringLen
        push eax
        pop eax
        mov lenENTRY, eax
        push strokaENTRY
        push offset L2
        call sunlight
```

```
        push eax
        pop eax
        mov resultENTRY, eax
        push resultENTRY
        call outStreamN
        push L3
        call factorial
        push eax
        push L4
        pop eax
        pop ebx
        mul ebx
        push eax
        call outStreamN
        push lenENTRY
        call outStreamN
        push L5
        jmp theend
theend:
        call ExitProcess
SOMETHINGWRONG:
        push offset null_division
        call outStreamW
        jmp konec
konec:
        push -1
        call ExitProcess
main ENDP
end main
```