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 1-40 01 01 10 «Программное обеспечение информационных технологий »

ПОЯСНИТЕЛЬНАЯ ЗАПИСКА
К КУРСОВОМУ ПРОЕКТУ НА ТЕМУ:

«Разработка компилятора PNA-2022»

Выполнил студент Песецкий Никита Андреевич
(Ф.И.О.)

Руководитель проекта ст. преп. Наркевич А.С.
(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.)

Консультант преп.-стажер Карпович М. Н.
(учен. степень, звание, должность, подпись, Ф.И.О.)

Нормоконтролер преп.-стажер Карпович М. Н.
(учен. степень, звание, должность, подпись, Ф.И.О.)

Курсовой проект защищен с оценкой _____

Минск 2022

Содержание

Введение

В данном курсовом проекте поставлена задача разработки собственного языка программирования и транслятора для него. Название языка – PNA-2022. Написание транслятора будет осуществляться на языке C++, при этом код на языке PNA-2022 будет транслироваться в язык ассемблера.

Задание на курсовой проект можно разделить на следующие задачи:

1. Разработка спецификации языка PNA-2022;
2. Разработка лексического анализатора;
3. Разработка синтаксического анализатора;
4. Разработка семантического анализатора;
5. Разбор арифметических выражений;
6. Разработка генератора кода;
7. Тестирование транслятора.

Глава 1. Спецификация языка программирования

1.1 Характеристика языка программирования

Язык программирования PNA-2022 является процедурным, строго типизируемым, транслируемым на язык assembler

1.2 Алфавит языка

Алфавит PNA-2022 состоит из символов, приведённых на рисунке 1.1

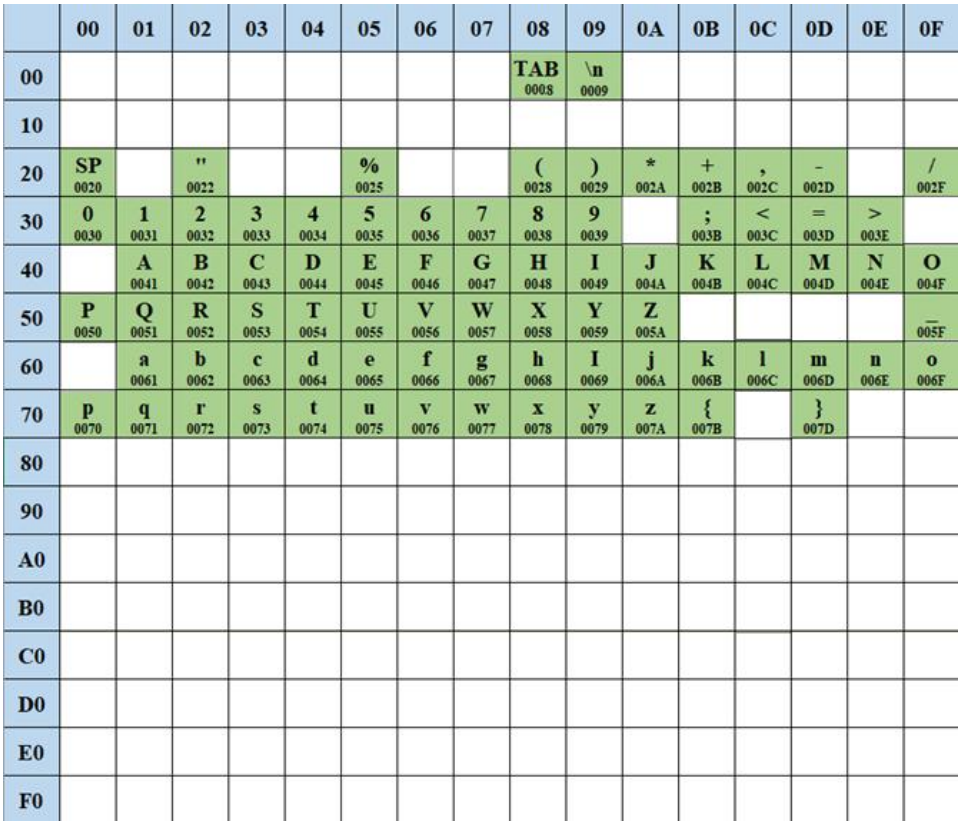

Рисунок 1.1 – Символы, разрешённые к использованию

Русские символы, а так же символы любых других языков в языке не используются.

1.3 Применяемые сепараторы

Сепараторы служат для разделения операций языка. Сепараторы, применяемые в языке программирования PNA-2022, приведены в таблице 1.1.

Таблица 1.1 – Применяемые сепараторы

Сепаратор	Назначение сепаратора
;	Разделитель инструкций
{ }	Программный блок

Продолжение таблицы 1.1.

Пробел	Допускается везде, кроме идентификаторов и ключевых слов. Служит для их разделения
,	Разделитель параметров в функции
( )	Параметры, приоритетность операций в выражениях

При несоблюдении правил использования сепараторов будет вызвана ошибка.

1.4 Применяемые кодировки

Кодировка, используемая для написания программ на языке PNA-2022 - стандартная кодировка Windows-1251, представленная на рисунке 1.2.

	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 1.2 – Используемая кодировка

В языке PNA-2022 будут использоваться не все символы данной кодировки.

1.5 Типы данных

Допускается использование фундаментальных типов данных определенных в таблице 1.2. Пользовательские типы данных не поддерживаются.

Таблица 1.2 – Фундаментальные типы данных

Тип данных	Описание
Int	Фундаментальный тип данных. Предусмотрен для объявления целочисленных данных (4 байта). Максимальное значение: 2147483647. Минимальное значение: -2147483648. Автоматически инициализируется нулевым значением. Возможные **арифметические** операции: **+** (бинарный) – операция сложения; **-** (бинарный) – операция вычитания; ***** (бинарный) – операция произведения; **/** (бинарный) – операция деления; **%** (бинарный) – операция деления нацело; **=** (бинарный) – оператор присваивания. Возможные варианты присваивания: Присваивание целочисленному идентификатору значения другого целочисленного идентификатора, целочисленного литерала или значения целочисленной функции. Возможные **логические** операции: **>** (бинарный) – оператор «больше»; **<** (бинарный) – оператор «меньше»;
Str	Фундаментальный тип данных. Предусмотрен для объявления строк. (1 символ – 1 байт). Автоматическая инициализация строкой нулевой длины. Максимальное количество символов в строке – 255. Первый байт – длина строки. Возможные операции: **=** (бинарный) – оператор присваивания. Возможные варианты присваивания: Присваивание строковому идентификатору значения другого строкового идентификатора, строкового литерала или значения строковой функции.

Используя данные типы можно покрыть большую часть задач.

1.6 Преобразование типов данных

Преобразование типов данных в языке PNA-2022 не предусмотрено. Попытка преобразования типов данных приведет к семантической ошибке.

1.7 Идентификаторы

Идентификаторы могут выступать в качестве имен функций, параметров, переменных. Зарезервированные идентификаторы предусмотрены. Идентификаторы не должны совпадать с ключевыми словами.

Имя идентификатора составляется по следующим правилам:

- задаётся таким регулярным выражением (^[A-z]\w*$)
- длина идентификатора не должна превышать 20 символов. При превышении максимально допустимой длины применяется усечение.

1.8 Литералы

Предусмотрены целочисленные и строковые литералы. Правила записи приведены в таблице 1.3.

Таблица 1.3 – Правила записи литералов

Тип литерала	Описание литерала
Целочисленный	PNA-2022 поддерживает представление литералов в следующих системах счисления: двоичная: [0, 1], начинается с префикса ‘b’; восьмеричная: [0..7], начинается с префикса ‘o’; десятичная: [0..9], без префикса; шестнадцатеричная: [0..9], [A..F], начинается с префикса ‘h’.
Строковый	Символы, заключенные в “…” (двойные кавычки), число которых не превышает 255.

Все литералы являются rvalue, т.е. одним из возможных типа данных,

1.9 Объявления данных и область видимости

Для объявления переменной указывается тип данных, далее используется ключевое слово **var**, после чего имя идентификатора.

Для объявления функций используется ключевое слово **def**, перед которым указывается тип функции, а после – идентификатор.

Область видимости реализована сверху-вниз. Все функции и процедуры имеют глобальную область видимости и могут быть объявлены только в глобальной области видимости. Переменные не могут быть глобальными. Любые идентификаторы должны быть объявлены до их использования.

1.10 Инициализация данных

Объектами-инициализаторами могут быть только идентификаторы или литералы. При объявлении предусмотрены значения по умолчанию: значение 0 для типа **int** и строка длины 0 (“”) для типа **str**.

1.11 Инструкции языка

В языке программирования PNA-2022 применяются инструкции, представленные в таблице 1.4.

Таблица 1.4 – Инструкции языка

Инструкция языка	Синтаксис
Объявление переменной	<тип данных> var <идентификатор>;
Присваивание	<идентификатор> = <литерал>|<выражение>|<идентификатор>;
Вызов функций	<идентификатор функции> ([параметр 1] [, параметр 2]);
Вывод данных в консоль	print(<литерал>|<идентификатор>);
Возврат из функции	return <литерал>|<идентификатор>;

При неправильном использовании инструкций языка ошибки будут выявлены на одном из этапов анализа кода.

1.12 Операции языка

Язык программирования PNA-2022 может выполнять арифметические операции, представленные в таблице 1.5.

Таблица 1.5. Приоритетности операций языка программирования PNA-2022

Операция	Арифметическое назначение	Приоритетность операции
()	Приоритетность операций	0
+	Сложение	1

Продолжение таблицы 7.1

-	Вычитание	1
*	Умножение	2
/	Деление	2
%	Взятие остатка от деления	2
>	Сравнение чисел	3
<	Сравнение чисел	3

За выполнения операции по приоритетам отвечает польская нотация.

1.13 Выражения и их вычисления

Выражения относятся только к целому типу.

Предусмотрены следующие правила составления выражений:

- выражения записываются до ввода сепаратора ‘;’;
- выражение может содержать вызов функций;
- рассматриваются слева направо;
- для изменения приоритета операция используются круглые скобки ();
- реализация выражений происходит с помощью обратной польской записи.

1.14 Программные конструкции языка

Программные конструкции представлены в таблице 1.7.

Таблица 1.6 – программные конструкции языка

Конструкция	Представление в языке

Главная функция	main
	{
	<инструкции языка>
	}

Функция	<тип данных> def <идентификатор>([<тип данных> arg <идентификатор>][, <тип данных> arg <идентификатор>])
	{
	<инструкции языка>
	return <идентификатор>|<литерал>
	}

Продолжение таблицы 1.6.

Цикл	repeat(<идентификатор1>|<литерал>
Условие	If([<идентификатор2>|<литерал>][<больше>|<меньше>] [<идентификатор2>|<литерал>]) then{<инструкции языка>} else{<инструкции языка>}

Эти конструкции встречаются в большинстве языках программирования.

1.15 Область видимости идентификаторов

Все идентификаторы, объявленные внутри функции, являются локальными. Параметры видны только внутри функции, в которой объявлены.

1.16 Семантические проверки

Перечень семантических проверок, предусмотренных языком, приведен в таблице 1.7.

Таблица 1.7 – Перечень семантических проверок

Номер	Правило
1	Наличие функции main
2	Усечение слишком длинных идентификаторов до 20 символов
3	Сначала осуществляется проверка на ключевые слова, а затем на идентификатор. Не допускаются идентификаторы, совпадающие с ключевыми словами
4	Нет повторяющихся наименований функций
5	Нет повторяющихся объявлений идентификаторов
6	Предварительное объявление, применяемых функций
7	Предварительное объявление, применяемых идентификаторов.
8	Соответствие типов формальных и фактических параметров при вызове функций
9	Усечение слишком длинного значения str-литерала
10	Округление слишком большого значения int-литерала

Продолжение таблицы 1.7.

11	Если ошибка возникает на этапе лексического анализа, синтаксический анализ не выполняется
12	Если 3 подряд фразы не разобраны, то работа транслятора останавливается
13	При возникновении ошибки в процессе синтаксического анализа, ошибочная фраза игнорируется (предполагается, что ее нет) и осуществляется попытка разбора следующей фразы. Граница фразы – точка с запятой.

Список ошибок был создан после изучения материала по возможном ошибкам в языке программирования

1.17 Распределение оперативной памяти на этапе выполнения

Транслированный код использует две области памяти. В сегмент констант заносятся все литералы. В сегмент данных заносятся переменные, параметры функций. Локальная область видимости в исходном коде определяется за счет использования переменной, хранящей имя родительского блока, что и обуславливает их локальность на уровне ассемблерного кода, все глобальные переменные имеют имя родительского блока “global”.

1.18 Стандартные библиотеки и их состав

Функции стандартной библиотеки и их описание представлено в таблице 1.8.

Таблица 1.8 – Стандартная библиотека языка PNA-2022

Имя функции	Возвращаемое значение	Принимаемые параметры	Описание
Adate	Integer	отсутствуют	Возвращает текущую дату в формате ddmmyyyy
Atime	Integer	отсутствуют	Возвращает текущее время в формате hhmm
Print	отсутствует	integer <идентификатор> | <целочисленный литерал>	Функция выводит на консоль <выражение>
Print	отсутствует	str <идентификатор> | <строковый литерал>	Функция выводит на консоль <выражение>

Функции были созданы по заданию из варианта курсового проекта.

1.19 Ввод и вывод данных

Ввод данных в языке PNA-2022 не предусмотрен. Вывод данных осуществляется с помощью ключевых слова “print”.

1.20 Точка входа

Точкой входа в программе является ключевое слово “main”. Точка входа не может отсутствовать или быть переопределена.

1.21 Препроцессор

Препроцессор в языке программирования PNA-2022 не предусмотрен.

1.22 Соглашения о вызовах

Используется соглашение stdcall, то есть все параметры передаются в стек справа налево, память освобождает вызываемым кодом.

1.23 Объектный код

Язык PNA-2022 транслируется в ассемблер.

1.24 Классификация сообщений транслятора

Классификация ошибок представлена в таблице 1.9.

Таблица 1.9 – Классификация ошибок

Номера ошибок	Характеристика
0 – 99	Системные ошибки
100 – 104	Ошибки входных параметров
105 – 109	Ошибки при открытии файла
110 – 119	Ошибки при чтении файла
120 – 140	Ошибки лексического анализа
600 – 610	Ошибки синтаксического анализа
700 – 720	Ошибки семантического анализа

Примеры ошибок будут представлены в следующих пунктах.

1.25 Контрольный пример

Контрольный пример демонстрирует главные особенности языка PNA-2022: его фундаментальные типы, основные структуры, функции, процедуры, использование функций статической библиотеки. Исходный код контрольного примера представлен в приложении А.

2 Структура транслятора

2.1 Компоненты транслятора, их назначение и принципы взаимодействия

В языке PNA-2022 исходный код транслируется в язык Assembler. Транслятор языка разделён на отдельные части, которые взаимодействуют между собой и выполняют отведённые им функции, которые представлены в пункте 2.1. Для того чтобы получить ассемблерный код, используется выходные данные работы лексического анализатора, а именно таблица лексем и таблица идентификаторов. Для указания выходных файлов используются входные параметры транслятора, которые описаны в таблице 2.1. Структура транслятора языка PNA-2022 приведена на рисунке 1.

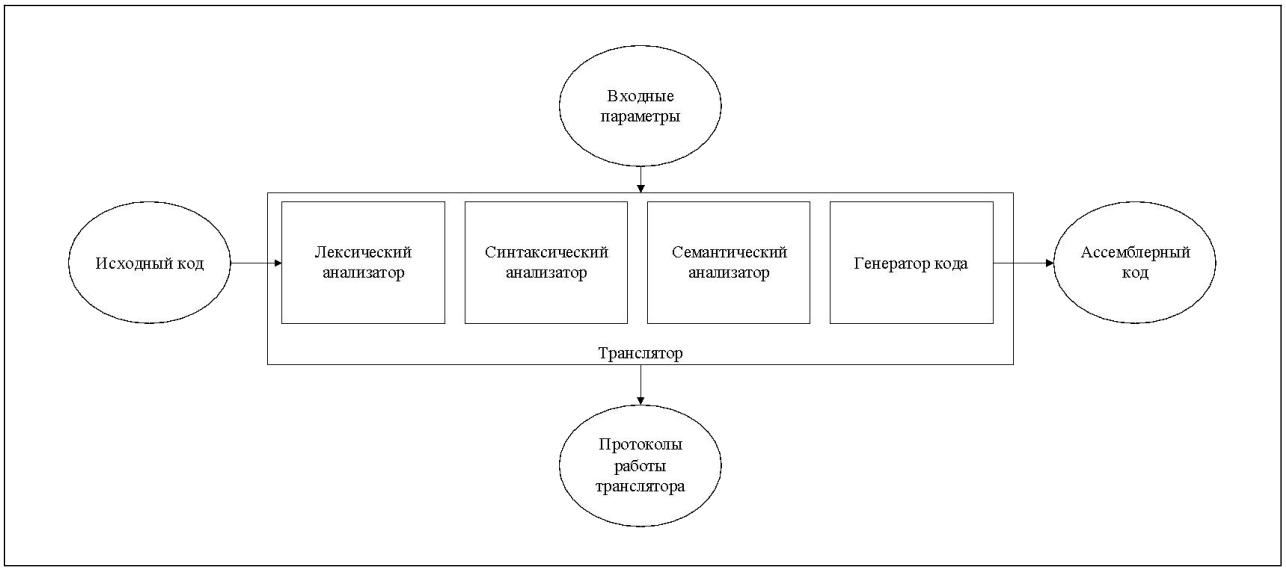

Рисунок 2.1 – Структура транслятора языка программирования PNA-2022

Первая стадия работы компилятора называется лексическим анализом, а программа, её реализующая, – лексическим анализатором (сканером). На вход лексического анализатора подаётся последовательность символов входного языка. Он производит предварительный разбор текста, преобразующий единый массив текстовых символов в массив отдельных слов (в теории компиляции вместо термина «слово» часто используют термин «токен»). Лексический анализатор преобразует исходный текст, заменяя лексические единицы их внутренним представлением – лексемами, для создания промежуточного представления исходной программы. Каждой лексеме сопоставляется ее тип и запись в таблице идентификаторов, в которой хранится дополнительная информация. Таблица лексем (ТЛ) и таблица идентификаторов (ТИ) являются входом для следующей фазы компилятора – синтаксического анализа (разбора, парсера).

Цели лексического анализатора:

− убрать все лишние пробелы;

− выполнить распознавание лексем;
− построить таблицу лексем и таблицу идентификаторов;
− Обработка лексических ошибок.

Синтаксический анализатор – часть компилятора, выполняющая синтаксический анализ, то есть проверку исходного кода на соответствие правилам грамматики. Входной информацией для синтаксического анализа является таблица лексем и таблица идентификаторов. Выходной информацией является дерево разбора

Семантический анализатор – часть транслятора, выполняющая семантический анализ, то есть проверку исходного кода на наличие ошибок, которые невозможно отследить при помощи регулярной и контекстно-свободной грамматики. Входными данными являются таблица лексем и идентификаторов.

Генератор кода – часть транслятора, выполняющая генерацию ассемблерного кода на основе полученных данных на предыдущих этапах трансляции. На вход генератора подаются таблица лексем и таблица идентификаторов, на основе которых генерируется файл с ассемблерным кодом.

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 PNA-2022

Входной параметр	Описание параметра	Значение по умолчанию
-in:<путь к in-файлу>	Файл с исходным кодом на языке PNA-2022 , имеющий расширение .txt	Не предусмотрено
-log:<путь к log-файлу>	Файл журнала для вывода протоколов работы программы.	Значение по умолчанию: <имя in-файла>.log
-out:<путь к out-файлу>	Выходной файл – результат работы транслятора. Содержит исходный код на языке асемблера.	Значение по умолчанию: <имя in-файла>.asm

Установка log и out файлов не обязательны. Значение по умолначию берётся из in файла.

2.3 Перечень протоколов, формируемых транслятором и их содержимое

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

Таблица 2.2 – Протоколы, формируемые транслятором языка PNA-2022

Формируемый протокол	Описание выходного протокола
Файл журнала, заданный параметром "-log:"	Файл с протоколом работы транслятора языка программирования PNA-2022. Содержит таблицу лексем и таблицу идентификаторов, протокол работы синтаксического анализатора и дерево разбора, полученные на этапе лексического и синтаксического анализа, а также результат работы алгоритма преобразования выражений к польской записи.
Выходной файл, заданный параметром "-out:"	Результат работы программы – файл, содержащий исходный код на языке ассемблера.

Исходный файл будет использоваться на этапе запуска программы, написанной на языке программирования PNA-2022.

3 Разработка лексического анализатора

3.1 Структура лексического анализатора

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

Примеры лексических единиц: идентификаторы, числа, символы операций, служебные слова и т.д. Лексический анализатор преобразует исходный текст, заменяя лексические единицы их внутренним представлением – лексемами, для создания промежуточного представления исходной программы.

Функции лексического анализатора:

1. удаление «пустых» символов и комментариев. Если «пустые» символы (пробелы, знаки табуляции и перехода на новую строку) и комментарии будут удалены лексическим анализатором, синтаксический анализатор никогда не столкнется с ними (альтернативный способ, состоящий в модификации грамматики для включения «пустых» символов и комментариев в синтаксис, достаточно сложен для реализации);
2. распознавание идентификаторов и ключевых слов;
3. распознавание констант;
4. распознавание разделителей и знаков операций.

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

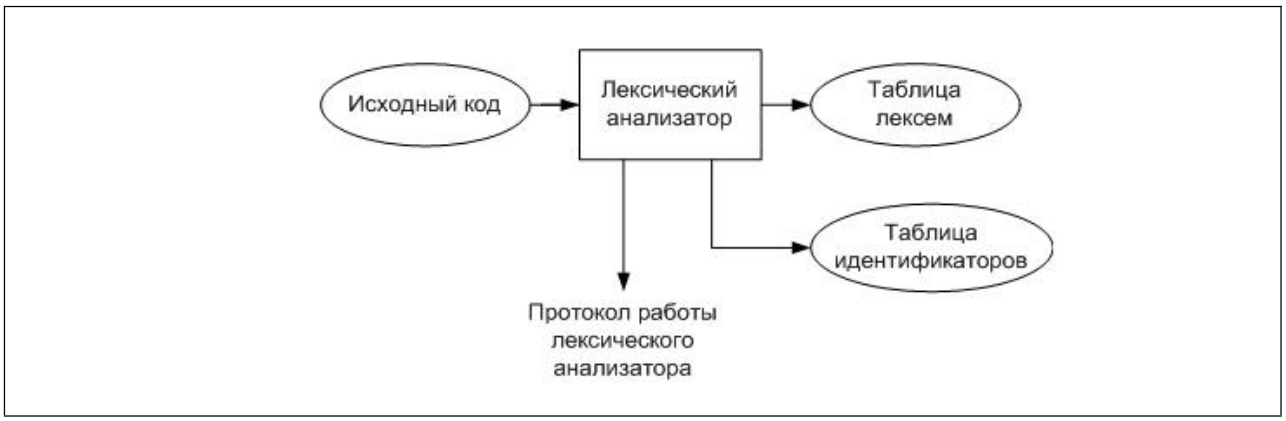

Рисунок 3.1 – Структура лексического анализатора

Каждой лексеме сопоставляется ее тип и запись в таблице идентификаторов, в которой хранится дополнительная информация.

3.2. Контроль входных символов

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

```
#define IN_CODE_TABLE {\
        IN::F, IN::F, IN::F, IN::F, IN::F, IN::F, IN::F, IN::F, IN::F, IN::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::F, IN::T, IN::F, IN::F, IN::V, IN::F, IN::F, IN::V, IN::V, IN::V, IN::V, IN::V, IN::V, IN::F, IN::V,\
        IN::N, IN::N, IN::N, IN::N, IN::N, IN::N, IN::N, IN::N, IN::N, IN::N, IN::F, IN::V, IN::V, IN::V, IN::V, 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::T,\
        IN::F, IN::T, IN::T, IN::T, IN::T, IN::T, IN::T, IN::T, IN::T, IN::T, IN::T, IN::T, IN::T, IN::T, IN::T, IN::T,\
        IN::T, IN::T, IN::T, IN::T, IN::T, IN::T, IN::T, IN::T, IN::T, IN::T, IN::T, IN::V, IN::F, IN::V, IN::F, IN::F,\
 \
        IN::F, IN::F, IN::F, IN::F, IN::F, IN::F, IN::F, IN::F, IN::F, IN::F, IN::F, IN::F, IN::F, IN::F, IN::F, IN::F,\
        IN::F, IN::F, IN::F, IN::F, IN::F, IN::F, IN::F, IN::F, IN::F, IN::F, IN::F, IN::F, IN::F, IN::F, IN::F, IN::F,\
        IN::F, IN::F, IN::F, IN::F, IN::F, IN::F, IN::F, IN::F, IN::F, IN::F, IN::F, IN::F, IN::F, IN::F, IN::F, IN::F,\
        IN::F, IN::F, IN::F, IN::F, IN::F, IN::F, IN::F, IN::F, IN::F, IN::F, IN::F, IN::F, IN::F, IN::F, IN::F, IN::F,\
        IN::F, IN::F, IN::F, IN::F, IN::F, IN::F, IN::F, IN::F, IN::F, IN::F, IN::F, IN::F, IN::F, IN::F, IN::F, IN::F,\
        IN::F, IN::F, IN::F, IN::F, IN::F, IN::F, IN::F, IN::F, IN::F, IN::F, IN::F, IN::F, IN::F, IN::F, IN::F, IN::F,\
        IN::F, IN::F, IN::F, IN::F, IN::F, IN::F, IN::F, IN::F, IN::F, IN::F, IN::F, IN::F, IN::F, IN::F, IN::F, IN::F,\
        IN::F, IN::F, IN::F, IN::F, 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

На данной таблице можно увидеть все доступные символы языка программирования.

Таблица 3.1 – Соответствие символов и их значений в таблице

Значение в таблице входных символов	Символы
Разрешенный	T
Запрещенный	F
Игнорируемый	I
Пробел, табуляция	S
Операторы	V
Цифры	N

Можно было разделить таблицу на запрещённые и разрешённые символы, но при проектировании было решено использовать другой подход

3.3 Удаление избыточных символов

Избыточными символами являются символы табуляции и пробелы.

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 3.2.

Таблица 3.2 – Соответствие токенов и сепараторов с лексемами

Токен	Лексема	Пояснение
int, string	T	Названия типов данных языка.
Идентификатор	I	Длина идентификатора – 20 символов.
Литерал	L	Литерал любого доступного типа.
def	D	Объявление функции.
var	V	Объявление переменной
arg	A	Объявление параметра функции
return	R	Выход из функции/процедуры.
main	M	Главная функция.
print	P	Вывод данных
if	?	Уловный оператор.
then	:	Истинная ветвь условного оператора.
else	!	Ложная ветвь условного оператора.
repeat	~	Указывает на начало тела цикла.
;	;	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.
+ - * / %	#	Знаки операций.

Продолжение таблицы 3.2.

>= <=	>= <=	Знаки логических операторов

Пример реализации таблицы лексем представлен в приложении Б.

Каждому выражению соответствует детерминированный конечный автомат, по которому происходит разбор данного выражения. На каждый автомат в массиве подаётся токен и с помощью регулярного выражения, соответствующего данному графу переходов, происходит разбор. Структура конечного автомата и пример графа перехода конечного автомата изображены на рисунках3.3 и 3.4 соответственно.

```
#pragma once
#include "stdafx.h"
#define MAX_LEN_CHAIN 100
namespace FST{
	struct RELATION{
		char symbol;
		short nnode;
		RELATION(
			char c,
			short ns
		);
	};
	struct NODE{
		short n_relation;
		RELATION* relations;
		NODE();
		NODE(
			short n,
			RELATION rel, ...
		);
	};
	struct FST{
		const char* string;
		char lexema;
		short position;
		short nstates;
		NODE* nodes;
		short* rstates;
		FST(
			const char* s,
			const char lex,
			short ns,
			NODE n, ...
		);
	};
	bool step(FST& fst, short*& rstates);
	bool execute(  FST& fst);};
```

Рисунок 3.3 Структура конечного автомата

В случае успешного разбора выражения оно записывается в таблицу лексем.

Если выражение является идентификатором или литералом, информация также заносится в таблицу идентификаторов.

```
#define FST_MAIN(string) FST::FST(string, LEX_MAIN, 5, \
FST::NODE(1, FST::RELATION('m', 1)), \
FST::NODE(1, FST::RELATION('a', 2)), \
FST::NODE(1, FST::RELATION('i', 3)), \
FST::NODE(1, FST::RELATION('n', 4)), \
FST::NODE())
```

Рисунок 3.4 – Пример реализации графа конечного автомата для токена main

Для других токенов так же созданы графы конечного автомата.

3.5 Основные структуры данных

Основными структурами данных лексического анализатора являются таблица лексем и таблица идентификаторов. Код C++ со структурой таблицы лексем представлен на рисунке 3.5. Код C++ со структурой таблицы идентификаторов представлен на рисунке 3.6.

```
namespace LT
{
	struct Entry
	{
		char lexema;
		char sign;
		int sn;
		int tn;
		int idxTI;

		Entry(char lexema, int sn, int tn, int idxTI);
		Entry(char lexema, char sign, int sn, int tn);
		Entry(char lexema, int sn, int tn);
		Entry(char lexema);
	};

	struct LexTable
	{
		int maxsize;
		int current_size;
		Entry** table;

		LexTable();
		LexTable(int size);
	};
	void Add(LexTable& lextable, Entry* entry);
	Entry GetEntry(LexTable& lextable, int n);
	void PrintLexTable(LexTable& lextable, const wchar_t* in);
	void Delete(LexTable& lextable);
}
```

Рисунок 3.5 – Структура таблицы лексем

Таблица лексем содержит номер лексемы, лексему (lexema), полученную при разборе, знак оператора (sign), номер строки в исходном коде (sn), номер токена (st), и номер в таблице идентификаторов, если

лексема является идентификатором (idxTI). Таблица идентификаторов содержит имя идентификатора (id), номер в таблице лексем (idxfirstLE), тип данных (iddatatype), тип идентификатора (idtype) и значение (или value).

```
	struct Entry
	{
		int idxfirstLE;
		char areaOfVisibility[ID_MAXSIZE]{};
		char id[ID_MAXSIZE]{};
		IDDATATYPE  iddatatype;
		IDTYPEidtype;
		union
		{
			int vint;
			struct
			{
				unsigned char len;
				char str[TI_STR_MAXSIZE];
			} vstr;
		}value;
		void SetVint(int num) { value.vint = num; };
		struct Param
		{
			int count;
			vector<IDDATATYPE> types;
		} params;
		string FullName;
		Entry(int idxfirstLE, string areaOfVisibility, const char* id,
IDDATATYPE iddatatype, IDTYPE idtype);
		Entry(int idxfirstLE ,IDDATATYPE iddatatype, IDTYPE idtype, char*
value);
	};
	struct IdTable
	{
		int maxsize;
		int current_size;
		Entry** table;

		IdTable(
			int size
			);
	};
```

Рисунок 3.6 – Структура таблицы идентификаторов

Для каждого идентификатора будет создан свой “Entry” и добавлен в таблицу.

3.6 Принцип обработки ошибок

Для обработки ошибок лексический анализатор использует таблицу с сообщениями. Перечень сообщений представлен на рисунке 3.7.

```
            ERROR_ENTRY(121, "Тип данных идентификатора не определен"),
            ERROR_ENTRY(122, "Превышен размер таблицы лексем"),
            ERROR_ENTRY(123, "Таблица лексем переполнена"),
            ERROR_ENTRY(124, "Попытка обращения к незаполненной строке таблицы
лексем"),
            ERROR_ENTRY(125, "Превышен размер лексемы"),
```

Рисунок 3.7 – Сообщения лексического анализатора.

При возникновении сообщения, лексический анализатор выбрасывает исключение – работа программы останавливается. Структура сообщений содержит информацию о номере сообщения, номер строки и позицию, где было вызвано сообщение в исходном коде, информацию об ошибке.

3.7 Структура и перечень сообщений лексического анализатора

Ошибки, возникающие в процессе трансляции программы, выводится на консоль, а также фиксируются в протокол, заданный входным параметрами. В случае возникновения ошибок происходит их протоколирование с номером ошибки и диагностическим сообщением.

3.8 Параметры лексического анализатора

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

3.9 Алгоритм лексического анализа

1. Проверяет входной поток символов программы на исходном языке на допустимость, удаляет лишние пробелы и добавляет сепаратор для вычисления номера строки для каждой лексемы;
2. Для выделенной части входного потока выполняется функция распознавания лексемы;
3. При успешном распознавании информация о выделенной лексеме заносится в таблицу лексем и таблицу идентификаторов, и алгоритм возвращается к первому этапу;
4. Формирует протокол работы;

5. При неуспешном распознавании выдается сообщение об ошибке.

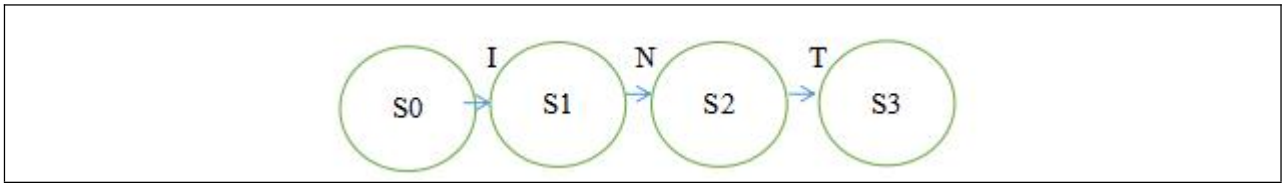

Рисунок 3.2 – Пример графа переходов для цепочки **str**

Распознавание цепочек основывается на работе конечных автоматов. Работу конечного автомата можно проиллюстрировать с помощью графа переходов. Пример графа для цепочки «**int**» представлен на рисунке 3.2, где S0 – начальное, а S3 – конечное состояние автомата.

3.10 Контрольный пример

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

4. Разработка синтаксического анализатора

4.1 Структура синтаксического анализатора

Синтаксический анализатор: часть компилятора, выполняющая синтаксический анализ, то есть исходный код проверяется на соответствие правилам грамматики.

Описание структуры синтаксического анализатора языка представлено на рисунке 4.1.

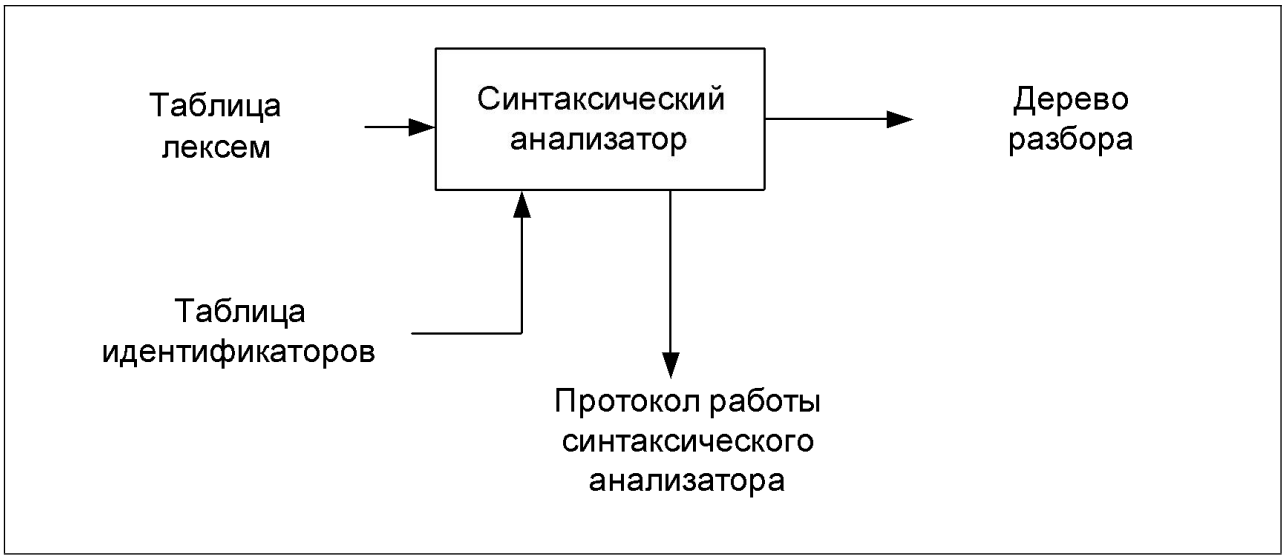

Рисунок 4.1 – Структура синтаксического анализатора.

Входной информацией для синтаксического анализа является таблица лексем и таблица идентификаторов. Выходной информацией– дерево разбора

4.2 Контекстно-свободная грамматика, описывающая синтаксис языка

В синтаксическом анализаторе транслятора языка PNA-2022 используется контекстно-свободная грамматика $G = \langle T, N, P, S \rangle$, где

Описание нетерминальных символов содержится в таблице 4.1.

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:

Таблица 4.1 – Правил переходов нетерминальных символов

Символ	Правила	Какие правила порождает
S	S-> m{N}S S-> tdi(F){NrU;}S S-> λ	Стартовые правила, описывающее общую структуру программы
N	N-> tY;N N-> i=E;N N-> p(Q);N N-> ~K{N}N N-> ?KJN N-> λ	Правила для операторов
U	U-> l U-> i	Только литерал или идентификатор
J	J-> :{N} J-> :{N}!{N}	Правила условного перехода
Y	Y-> vi Y-> vi=U Y-> vi=i(W) Y-> vi=i( ) Y-> vi=U#U	Правило определения переменной
K	K-> (U<U) K-> (U>U)	Правила определения условия перехода
E	E-> iM E-> lM E-> (E)M E-> i(W)M	Правила выражений
M	M-> #E M-> λ	Правила арифметический операторов
F	F-> tP F-> tP,F	Правила определения параметров функции
P	P-> ai	Правило определения параметров функции
W	W-> l W-> i W-> l,W W-> i,W W->#l,W	Правила вызова функции с параметрами

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

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

4.3 Построение конечного магазинного автомата

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

Таблица 4.2 – Компоненты магазинного автомата.

Компонента	Определение	Описание
Q	Множество состояний автомата	Состояние автомата представляет из себя структуру, содержащую позицию на входной ленте, номера текущего правила и цепочки и стек автомата
V	Алфавит входных символов	Алфавит представляет из себя множества терминальных и нетерминальных символов, описание которых содержится в таблица 3.1 и 4.1.
Z	Алфавит специальных магазинных символов	Алфавит магазинных символов содержит стартовый символ и маркер дна стека (представляет из себя символ $)
δ	Функция переходов автомата	Функция представляет из себя множество правил грамматики, описанных в таблице 4.1.
q_0	Начальное состояние автомата	Состояние, которое приобретает автомат в начале своей работы. Представляется в виде стартового правила грамматики
z_0	Начальное состояние магазина автомата	Символ маркера дна стека $
F	Множество конечных состояний	Конечные состояние заставляют автомат прекратить свою работу. Конечным состоянием является пустой магазин автомата и совпадение позиции на входной ленте автомата с размером ленты

Магазинный автомат используется для анализа итогового выражения.

4.4 Основные структуры данных

Основные структуры данных синтаксического анализатора представляются в виде структуры магазинного конечного автомата, выполняющего разбор исходной ленты, и структуры грамматики Грейбах, описывающей синтаксические правила языка PNA-2022. Данные структуры в приложении В.

4.5 Описание алгоритма синтаксического разбора

Принцип работы автомата, следующий:

1. В магазин записывается стартовый символ;
2. На основе полученных ранее таблиц формируется входная лента;
3. Запускается автомат;
4. Выбирается цепочка, соответствующая нетерминальному символу, записывается в магазин 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 нетерминала;
6. Если в магазине встретился нетерминал, переходим к пункту 4;
7. Если наш символ достиг дна стека, и лента в этот момент пуста, то синтаксический анализ выполнен успешно. Иначе генерируется исключение.

4.6 Структура и перечень сообщений синтаксического анализатора

Перечень сообщений синтаксического анализатора представлен на рисунке 4.3.

```
            ERROR_ENTRY(600, "Неверная структура программы"),
            ERROR_ENTRY(601, "Ошибочный оператор"),
            ERROR_ENTRY(602, "Ошибка в выражении"),
            ERROR_ENTRY(603, "Ошибка в параметрах при определении функции"),
            ERROR_ENTRY(604, "Ошибка в параметрах при вызове функции"),
            ERROR_ENTRY(605, "Ошибка: только литерал или идентификатор"),
            ERROR_ENTRY(606, "Ошибка при условном переходе"),
            ERROR_ENTRY(607, "Ошибка при определении переменной"),
            ERROR_ENTRY(608, "Ошибка при определении условия перехода"),
        ERROR_ENTRY_NODEF(609),
```

Рисунок 4.3 – Сообщения синтаксического анализатора

Программа прекращает работу при нахождении первой ошибки.

4.7. Параметры синтаксического анализатора и режимы его работы

Входной информацией для синтаксического анализатора является таблица лексем и идентификаторов. Кроме того, используется описание грамматики в форме Грейбах. Результаты работы лексического разбора, а именно дерево разбора и протокол работы автомата с магазинной памятью выводятся в журнал работы программы.

4.8. Принцип обработки ошибок

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

4.9. Контрольный пример

Результаты работы лексического разбора, а именно дерево разбора и протокол работы автомата с магазинной памятью приведены в приложении В.

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. Общая структура обособленно работающего (не параллельно с лексическим анализом) семантического анализатора представлена на рисунке 5.1.

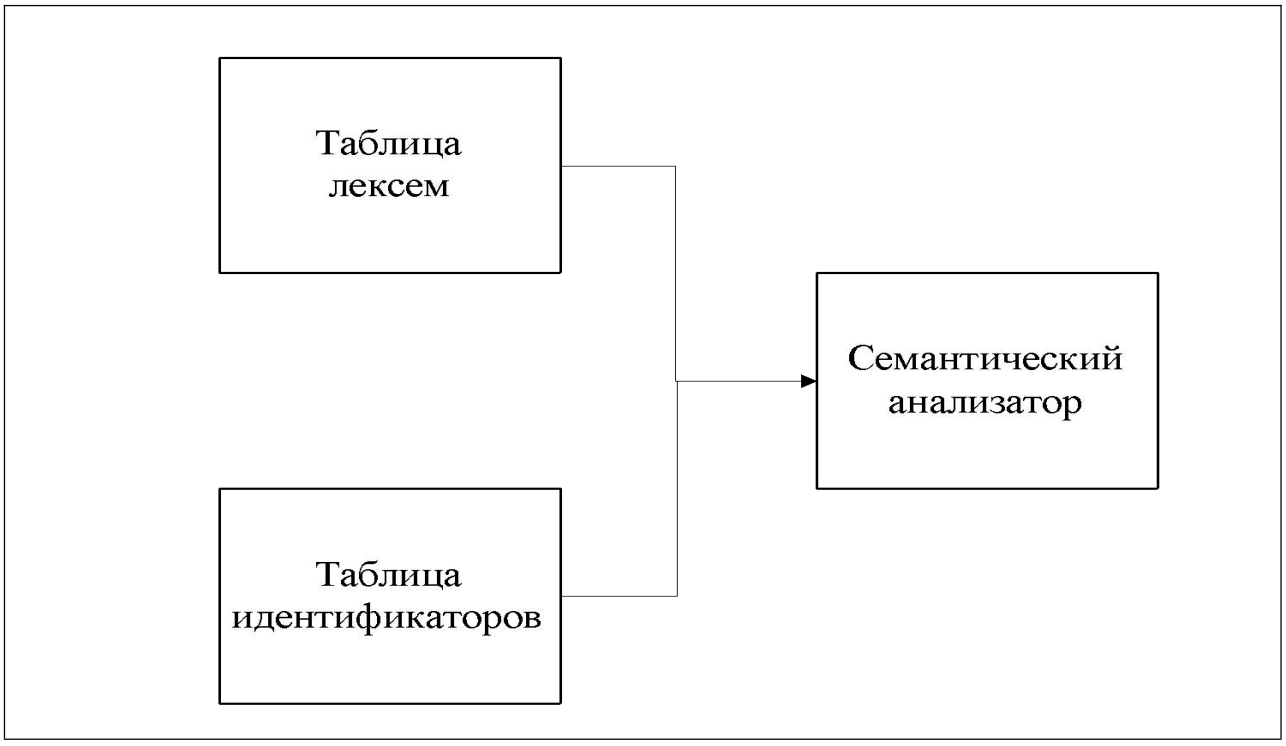

Рисунок 5.1 – Структура семантического анализатора

Некоторые проверки (такие как проверка на единственность точки входа, проверка на предварительное объявление переменной) осуществляются в процессе лексического анализа.

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, "Деление на 0"),
ERROR_ENTRY(701, "Типы данных в выражении не совпадают"),
ERROR_ENTRY(702, "Недопустимое строковое выражение справа от знака \'=\'"),
ERROR_ENTRY(703, "Тип функции и возвращаемое значение не совпадают"),
ERROR_ENTRY(704, "Несовпадение типов передаваемых параметров"),
ERROR_ENTRY(705, "Кол-во ожидаемых функцией и передаваемых параметров не
совпадают"),
ERROR_ENTRY(706, "Неверное условное выражение"),
```

Рисунке 5.2 – Перечень сообщений семантического анализатора

5.4 Принцип обработки ошибок

Ошибки, возникающие в процессе трансляции программы, выводиться на консоль, а также фиксируются в протокол, заданный входным параметрами. В случае возникновения ошибок происходит их протоколирование с номером ошибки и диагностическим сообщением.

5.5 Контрольный пример

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

Таблица 5.1 – Примеры диагностики ошибок

Исходный код	Текст сообщения

main{
int var x;
x = 1 + “test1”;
}
	Ошибка на этапе Семантического анализатора
	Ошибка 701: Типы данных в выражении не совпадают, строка 3

main{
int var x;
x = 5 / 0;
}
	Ошибка на этапе Семантического анализатора
	Ошибка 700: Деление на 0, строка 3, лексема 5

main{
str var y;
y = 1 + 2;
}
	Ошибка на этапе Семантического анализатора
	Ошибка 702: Недопустимое строковое выражение справа от знака '=', строка 3

Если ошибки не будут найденный, программа перейдёт к следующему этапу.

6. Вычисление выражений

6.1 Выражения, допускаемые языком

В языке PNA-2022 допускаются вычисления выражений целочисленного типа данных с поддержкой вызова функций внутри выражений. Приоритет операций представлен на таблице 6.1.

Таблица 6.1 – Приоритеты операций

Операция	Значение приоритета
%	4
/	4
*	4
-	3
+	3
,	2
( )	1

Вычисление выражений сильно упрощает написание кода, с помощью этого сложные вычисления можно записать в одну строку.

6.2 Польская запись и принцип её построения

Все выражения языка PNA-2022 преобразовываются к обратной польской записи.

Польская запись – это альтернативный способ записи арифметических выражений, преимущество которого состоит в отсутствии скобок. Существует два типа польской записи: прямая и обратная, также известные как префиксная и постфиксная. Отличие их от классического, инфиксного способа заключается в том, что знаки операций пишутся не между, а, соответственно, до или после аргументов. Алгоритм построения польской записи:

исходная строка: выражение;

1. результирующая строка: польская запись;
2. стек: пустой;
3. исходная строка просматривается слева направо;
4. операнды переносятся в результирующую строку;
5. операция записывается в стек, если стек пуст;
6. операция выталкивает все операции с большим или равным приоритетом в результирующую строку;
7. отрывающая скобка помещается в стек;
8. закрывающая скобка выталкивает все операции до открывающей скобки, после чего обе скобки уничтожаются.

6.3 Программная реализация обработки выражений

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

6.4 Контрольный пример

Пример преобразования выражений из контрольных примеров к обратной польской записи представлен в таблице 6.2.

Таблица 6.2 – Преобразование выражений

Выражение	Обратная польская запись для выражения
L	l
i#l	il#
i#l	il#

Далее в файле вместо польская инверсная запись будет использоваться ПОЛИЗ.

7. Генерация кода

7.1 Структура генератора кода

В языке PNA-2022 генерация кода является заключительным этапом трансляции. Генератор принимает на вход таблицы лексем и идентификаторов, полученные в результате лексического анализа. Структура генератора кода PNA-2022 представлена на рисунке 7.1.

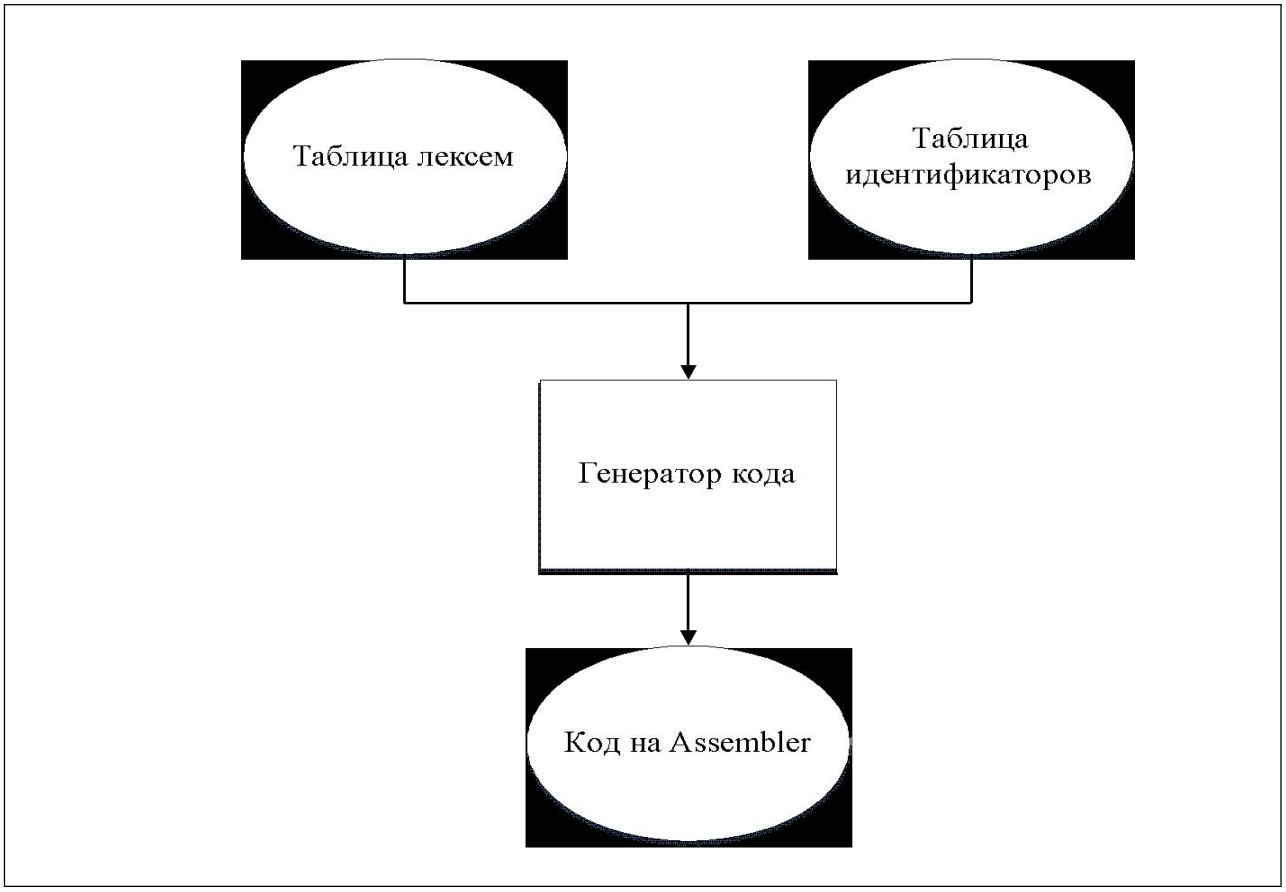

Рисунок 7.1 – Структура генератора кода

В соответствии с таблицей лексем строится выходной файл на языке ассемблера, который будет являться результатом работы транслятора. В случае возникновения ошибок генерация кода не будет осуществляться.

7.2 Представление типов данных в оперативной памяти

Элементы таблицы идентификаторов расположены в сегментах .data и .const языка ассемблера. Соответствия между типами данных идентификаторов на языке PNA-2022 и на языке ассемблера приведены в таблице 7.1.

Таблица 7.1 – Соответствия типов идентификаторов языка PNA-2022 и языка ассемблера

Тип идентификатора на языке PNA-2022	Тип идентификатора на языке ассемблера	Пояснение
str	Dword	Хранит указатель на начало строки. Строка должна завершаться нулевым символом.
int	Sdword	Хранит целочисленный тип.

На данный момент это все виды данных, присутствующие в языке.

7.3 Статическая библиотека

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

Таблица 7.3 – Функции статической библиотеки

Функция	Назначение
void print_str(char* str)	Вывод на консоль строки str
void print_int(int num)	Вывод на консоль целочисленной переменной num
int atime()	Возвращает текущую дату
int adate()	Возвращает текущее время

Объявление функций статической библиотеки генерируется автоматически в коде ассемблера.

7.4 Особенности алгоритма генерации кода

В языке PNA-2022 генерация кода строится на основе таблиц лексем и идентификаторов.

Общая схема работы генератора кода представлена на рисунке 7.2.

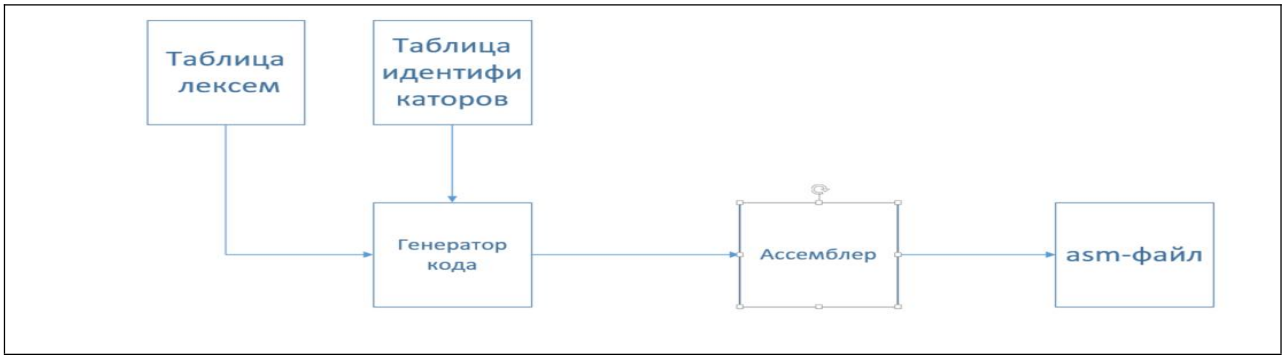

Рисунок 7.2 – Структура генератора кода

Во время генерации создаётся файл формата .asm в который записывается наша программа на языке ассемблера.

7.5 Входные параметры генератора кода

На вход генератору кода поступают таблицы лексем и идентификаторов исходного код программы на языке PNA-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 приведён на рисунок 7.2.

```
1972
17122022
2220
2747
5
```

Рисунок 7.2 – Результат работы программы на языке PNA-2022

Данный результат можно увидеть, запустив asm файл, собранный по приложению А.

8. Тестирование транслятора

8.1 Тестирование проверки на допустимость символов

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

Таблица 8.1 – Тестирование проверки на допустимость символов

Исходный код	Диагностическое сообщение
main { ~ }	Ошибка при чтении из входного файла Ошибка 110: Недопустимый символ в исходном файле (-in), строка 1, столбец 8

main
{
print("test);
}
	Ошибка при чтении из входного файла
	Ошибка 111: Нет закрывающей кавычки (-in), строка 3, столбец 15

При использовании игнорируемых символов ошибок не будет. Они будут игнорироваться на этапе чтения файла.

8.2 Тестирование лексического анализатора

На этапе лексического анализа в языке PNA-2022 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
{
test123
}
	Ошибка на этапе лексического анализатора
	Ошибка 120: Нераспознанная лексема, строка 3, лексема 1

main
{
var test;
}
	Ошибка на этапе лексического анализатора
	Ошибка121: Тип данных идентификатора не определен, строка 3, лексема 2

Продолжение таблицы 8.2 – Тестирование лексического анализатора

main {}
main
{
int var test;
}
	Ошибка 134: Обнаружено несколько точек входа main

int def FindMaxLen(str arg x, str arg y){}	Ошибка 133: Отсутствует точка входа main

main
{
int test;
}
	Ошибка на этапе лексического анализатора
	Ошибка 132: Тип идентификатора не определен, строка 3, лексема 2

main {
int var test;
int var test;
}
	Ошибка на этапе лексического анализатора
	Ошибка 131: Переопределение, строка 4, лексема 3

Лексический анализ это второй этап обработки программы написанной на языке PNA-2022.

8.3 Тестирование синтаксического анализатора

На этапе синтаксического анализа в языке PNA-2022 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

def main
{
int var test;
}
	600: строка 1, Неверная структура программы

Main
{
return 1;
}
	601: строка 4, Ошибочный оператор

Main
{
int var test;
test = 1 + - 2;
}
	602: строка 4, Ошибка в выражении

Продолжение таблицы 8.3 – Тестирование синтаксического анализатора

main
{
if (1 > 0)
then
}
	606: строка 4, Ошибка при условном переходе

main
{
int str var test;
}
	607: строка 3, Ошибка при определении переменной

main
{
if {1 > 0)
then {}
}
	608: строка 3, Ошибка при определении условия перехода

int def FindMax(int arg x)
{ return 1;}
main
{
int var res;
res = FindMax(1,);
}
	604: строка 6, Ошибка в параметрах при вызове функции

int def FindMaxLen(str arg x,) {} main {}	603: строка 1, Ошибка в параметрах при определении функции

Синтаксический анализ это третий этап обработки программы написанной на языке PNA-2022.

8.4 Тестирование семантического анализатора

Семантический анализ в языке PNA-2022 содержит множество проверок по семантическим правилам, описанным в пункте 1.16. Итоги тестирования семантического анализатора на корректное обнаружение семантических ошибок приведены в таблице 8.4.

Таблица 8.4 – Тестирование семантического анализатора

Исходный код	Текст сообщения

main{
int var x;
x = 1 + “test1”;
}
	Ошибка на этапе Семантического анализатора
	Ошибка 701: Типы данных в выражении не совпадают, строка 3

Продолжение таблицы 8.4 – Тестирование семантического анализатора

int def Test(int arg x)
{return 0;}
main
{
int var temp;
temp = Test("test");
}
	Ошибка на этапе Семантического анализатора
	Ошибка 704: Несовпадение типов передаваемых параметров, строка 6, лексема 5

int def Test(int arg x)
{return 0;}
main
{
int var temp;
temp = Test(1, 2);
}
	Ошибка на этапе Семантического анализатора
	Ошибка 705: Кол-во ожидаемых функцией и передаваемых параметров не совпадают, строка 6

main
{
if ("test1" > "test2")
then{}
}
	Ошибка на этапе Семантического анализатора
	Ошибка 706: Неверное условное выражение, строка 3

str def Test(int arg x)
{
return 0;
}
main {}
	Ошибка на этапе Семантического анализатора
	Ошибка 703: Тип функции и возвращаемое значение не совпадают, строка 3

main{
str var y;
y = "test2" + “test3”;
}
	Ошибка на этапе Семантического анализатора
	Ошибка 702: Недопустимое строковое выражение справа от знака '=', строка 3

main{
int var x;
x = 5 / 0;
}
	Ошибка на этапе Семантического анализатора
	Ошибка 700: Деление на 0, строка 3, лексема 5

Семантический анализатор это четвертый этап обработки программы написанной на языке PNA-2022

Заключение

В ходе выполнения курсовой работы был разработан транслятор и генератор кода для языка программирования PNA-2022 со всеми необходимыми компонентами. Таким образом, были выполнены основные задачи данной курсовой работы:

Сформулирована спецификация языка PNA-2022;

1. Разработаны конечные автоматы и важные алгоритмы на их основе для эффективной работы лексического анализатора;
2. Осуществлена программная реализация лексического анализатора, распознающего допустимые цепочки спроектированного языка;
3. Разработана контекстно-свободная, приведённая к нормальной форме Грейбах, грамматика для описания синтаксически верных конструкций языка;
4. Осуществлена программная реализация синтаксического анализатора;
5. Разработан семантический анализатор, осуществляющий проверку используемых инструкций на соответствие логическим правилам;
6. Разработан транслятор кода на язык ассемблера;
7. Проведено тестирование всех вышеперечисленных компонентов.

Окончательная версия языка PNA-2022 включает:

1. 2 типа данных;
2. Поддержка оператора вывода;
3. Возможность вызова функций стандартной библиотеки;
4. Наличие 5 арифметических операторов для вычисления выражений;
5. Поддержка функций, процедур, операторов цикла и условия;
6. Структурированная и классифицированная система для обработки ошибок пользователя.

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

Список использованных источников

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

2. Вирт, Н. Построение компилятора / Н. Вирт.– Москва : ДМК, 2010. – 194 с.

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

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

5. Макконнелл, С. Совершенный код. Мастер класс / С. Макконнелл.– 2010 – 889 с.

Приложение А

Листинг 1 – исходный код программы на языке PNA-2022

```
int def func(int arg ccc)
{
     if(5 > ccc)
     then
     {
          print(5);
     }
     else
     {
          print(ccc);
     }
     return 0;
}

main
{
     int var callf = func(4);
}}
```

Приложение Б

Листинг 1. Таблица идентификаторов контрольного примера

ПЕРЕМЕННЫЕ

Индекс LE	Область видимости	Идентификатор	Тип Идентификатора	Тип данных	Значение	Длина строки
6	func	ccc	P	INT	5	-
39	main	callf	V	INT	0	-

ФУНКЦИИ

Индекс LE	Область видимости	Идентификатор	Тип данных возврата
1	global	adate	INT
1	global	atime	INT
2	global	func	INT

ЛИТЕРАЛЫ

Индекс LE	Идентификатор	Тип данных	Значение	Длина строки
11	L0	INT	5	-
32	L1	INT	0	-
43	L2	INT	4	-

Листинг 2. Таблица лексем после контрольного примера

```
0   : S->tdi(F){NrU;}S
4   : F->tP
5   : P->ai
9   : N->?KJN
10  : K->(U>U)
11  : U->l
13  : U->i
15  : J->:{N}!{N}
17  : N->p(W);N
19  : W->l
22  : N->
25  : N->p(W);N
27  : W->i
30  : N->
31  : N->
32  : U->l
35  : S->m{N}S
37  : N->tY;N
38  : Y->vi=i(W)
43  : W->l
46  : N->
47  : S->   ¶
```

Приложение В

Листинг 1. Грамматика языка PNA-2022

```
namespace GRB
{
	#define NS(n) Rule::Chain::N(n)
	#define TS(n) Rule::Chain::T(n)

	Greibach greibach(NS('S'), TS('$'), 11,
		Rule(NS('S'), GRB_ERROR_SERIES + 0, //Неверная
структура программы
			3,	// m{N}S | tdi(F){NrU;}S | пустой переход
```

```
                Rule::Chain(5, TS('m'), TS('{'), NS('N'), TS('}'),
NS('S')),
                Rule::Chain(13, TS('t'), TS('d'), TS('i'),
TS('('), NS('F'), TS(')'), TS('{'), NS('N'), TS('r'), NS('U'),
TS(';'), TS('}'), NS('S')),
                Rule::Chain()
            ),
            Rule(NS('N'), GRB_ERROR_SERIES + 1,     //ошибочный
оператор
                6,   // tY;N |  i=E;N | p(W);N | ~K{N}N | ?KJN |
пустой переход
                Rule::Chain(4, TS('t'), NS('Y'), TS(';'),
NS('N')),
                Rule::Chain(5, TS('i'), TS('='), NS('E'), TS(';'),
NS('N')),
                Rule::Chain(6, TS('p'), TS('('), NS('W'),
TS(')'), TS(';'), NS('N')),
                Rule::Chain(6, TS('~'), NS('K'), TS('{'), NS('N'),
TS('}'), NS('N')),
                Rule::Chain(4, TS('?'), NS('K'), NS('J'),
NS('N')),
                Rule::Chain()
            ),
            Rule(NS('U'), GRB_ERROR_SERIES + 5,     // Только
литерал или идентификатор
                2,   // l | i
                Rule::Chain(1, TS('l')),
                Rule::Chain(1, TS('i'))
            ),
            Rule(NS('J'), GRB_ERROR_SERIES + 6,     // Ошибка при
условном переходе
                2,   // :{N} | :{N}!{N}
                Rule::Chain(4, TS(':'), TS('{'), NS('N'),
TS('}')),
                Rule::Chain(8, TS(':'), TS('{'), NS('N'), TS('}'),
TS('!'), TS('{'), NS('N'), TS('}'))
            ),
            Rule(NS('Y'), GRB_ERROR_SERIES + 7,     // Ошибка при
определении переменной
                5,   // vi | vi=U | vi=i(W) | vi=U#U
                Rule::Chain(2, TS('v'), TS('i')),
                Rule::Chain(4, TS('v'), TS('i'), TS('='),
NS('U')), // идентификатор равен U
                Rule::Chain(7, TS('v'), TS('i'), TS('='), TS('i'),
TS('('), NS('W'), TS(')')), // идентификатор равен функция с
```

```
параметрами
                Rule::Chain(6, TS('v'), TS('i'), TS('='), TS('i'),
TS('('), TS(')')), // идентификатор равен функция без параметров
                Rule::Chain(6, TS('v'), TS('i'), TS('='), NS('U'),
TS('#'), NS('U')) // идентификатор равен выражение
            ),
            Rule(NS('K'), GRB_ERROR_SERIES + 8,     // Ошибка при
определении условия перехода
                2,   // (U<U) | (U>U)
                Rule::Chain(5, TS('('), NS('U'), TS('<'), NS('U'),
TS(')')),
                Rule::Chain(5, TS('('), NS('U'), TS('>'), NS('U'),
TS(')'))
            ),
            Rule(NS('E'), GRB_ERROR_SERIES + 2,          // Ошибка
в выражении
                4, // iM | lM | (E)M | i(W)M
                Rule::Chain(2, TS('i'), NS('M')),
                Rule::Chain(2, TS('l'), NS('M')),
                Rule::Chain(4, TS('('), NS('E'), TS(')'),
NS('M')),
                Rule::Chain(5, TS('i'), TS('('), NS('W'), TS(')'),
NS('M'))
            ),
            Rule(NS('M'), GRB_ERROR_SERIES + 2,          //
     Ошибка в выражении
                2,   // #E | пустой переход
                Rule::Chain(2, TS('#'), NS('E')),
                Rule::Chain()
            ),
            Rule(NS('F'), GRB_ERROR_SERIES + 3,          // Ошибка
в параметрах при определении функции
                2, // tP | tP,F
                Rule::Chain(2, TS('t'), NS('P')),
                Rule::Chain(4, TS('t'), NS('P'), TS(','), NS('F'))
            ),
            Rule(NS('P'), GRB_ERROR_SERIES + 3,          // Ошибка
в параметрах при определении функции
                1, // ai
                Rule::Chain(2, TS('a'), TS('i'))
            ),
            Rule(NS('W'), GRB_ERROR_SERIES + 4,          // Ошибка
в параметрах при вызове функции
                4, // i | l | i,W | l,W | #l,W
                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::Chain(2, TS('#'), TS('l'), TS(','), NS('W'))
          )
     );
```

Листинг 2. Структура магазинного автомата

```
namespace MFST
{
     struct MFSTState    // состояние автомата (для сохранения)
     {
          short posInLent;          // позиция на ленте
          short nRule;              // номер текущего правила
          short nRuleChain;         // номер текущей цепочки,
текущего правила
          MFSTSTSTACK st;     // стек автомата

          MFSTState();
          MFSTState(short posInLent,          // позиция на ленте
               MFSTSTSTACK a_steck,           // стек автомата
               short currentChain);           // номер текущей
цепочки, текущего правила

          MFSTState(short posInLent,          // позиция на ленте
               MFSTSTSTACK a_stack,           // стек автомата
               short currentRule,             // номер текущего
правила
               short currentChain);           // номер текущей
цепочки, текущего правила

     };

     struct MFST // магазинный автомат
     {
          enum RC_STEP //код возврата 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 походящая
цепочка правила (ошибка в исходном коде)
               NS_ERROR,       // неизвесный нетерминальный символ
```

```
грамматики
                TS_OK,                  // тек. символ ленты ==
вершине стека, продвинулась лента, pop стека
                TS_NOK,                 // тек. символ ленты !=
вершине стека, восстановленно состояние
                LENTA_END,              // теущая позиция ленты >=
lenta_size
                SURPRISE        // неожиданный код возврата (ошибка
в step)
          };

          struct MFST_Diagnosis    // диагностика
          {
                short     posInLent;          // позиция на ленте
                RC_STEP   rc_step;            // код завершения
шага
                short     ruleNum;            // номер правила
                short     nrule_chain;        // номер цепочки
правила
                MFST_Diagnosis();
                MFST_Diagnosis(short posInLent, RC_STEP rc_step,
short ruleNum, short ruleChainNum);
          } diagnosis[MFST_DIAGN_NUMBER]; // последние самые
глубокие сообщения

          GRBALPHABET* lenta;                     //
перекодированная (TS/NS) лента (из LEX)
          short currentPosInLent;                 // текущая
позиция на ленте
          short currentRule;                      // номер
текущего правила
          short currentRuleChain;                 // номер
текущей цепочки, текущего правила
          short lenta_size;                       // размер ленты
          GRB::Greibach grebach;                  // грамматика
Грейбах
          LT::LexTable lexTable;
          MFSTSTSTACK st;                              // стек
автомата
          std::stack<MFSTState> storestate;  // стек для
сохранения состояний

          MFST();
          MFST(const LT::LexTable& lexTable,GRB::Greibach
grebach);
```

```
          char* getCSt(char* buf);
     //получить содержиое стека
          char* getCLenta(char* buf, short pos, short n = 25);
     //лента: n символов, начиная с pos
          char* getDiagnosis(short n, char* buf, int&
error_id_GRB);                    //получить n-ую строку
диагностики или '\0'

          bool savestate(std::ostream* stream);   //сохранить
состояние автомата
          bool reseteстate(std::ostream* stream); //восстановить
состояние автомата
          bool push_chain(GRB::Rule::Chain chain);

          RC_STEP step(std::ostream* stream);
     //выполнить шаг автомата
          bool start(std::ostream* stream);       //запустить
автомат
          bool savedDiagnosis(RC_STEP prc_step);

          void printRules(std::ostream* stream); //вывести
последовательность правил

          struct Deducation        // вывод
          {
               short stepsCount;              // количество шагов
в выводе
               short* nRules;                 // номер правила
грамматики
               short* nChainsOfRules;         // номер цепочек
правил грамматики

               Deducation()
               {
                    this->stepsCount = 0;
                    this->nRules = 0;
                    this->nChainsOfRules = 0;
               }
          }deducation;

          bool saveoutputTree();              // сохранить дерево
вывода
     };
}
```

Листинг 3. Структура грамматики Грейбах

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

		struct Chain			// цепочка (правая часть правила)
		{
			short size;				// длина цепочки
			GRBALPHABET* nt;	// цепочка терминалов (>0) и нетерминалов (<0)

			Chain()
			{
				size = 0;
				nt = new GRBALPHABET[1]{};
			};

			Chain(short symbolCount, // количество символов в цепочке
				GRBALPHABET s, ...);	// символы (терминал или нетерминал)

			char* getCChain(char* b);		//получить правую сторону правила
			static GRBALPHABET T(char t) { return GRBALPHABET(t); }		// терминал
			static GRBALPHABET N(char n) { return -GRBALPHABET(n); }	// нетерминал
			static bool isT(GRBALPHABET s) { return s > 0; }
		// терминал?
			static bool isN(GRBALPHABET s) { return !isT(s); }
			// нетерминал?
			static char alphabet_to_char(GRBALPHABET s) { return (isT(s) ? char(s) : char(-s)); }	// CRBALPHABET->char
		}*chains;		// массив цепочек - правых частей правила
```

```
          Rule()
          {
               nn = 0;
               size = 0;
          }
          Rule(GRBALPHABET pnn,    // нетерминал (<0)
               int iderror,        // идентификатор
диагностического сообщения (Error)
               short psize,        // количество цепочек - правых
частей правила
               Chain c, ...);      // множество цепочек - правых
частей правила

          char* getCRule(               // получить правило в
виде N->цепочка (для распечатки)
               char* b,            // буфер
               short nchain);      // номер цепочки (правой
части) в правиле

          short getNextChain(           // получить следующую за
j подходящую цепочку, вернуть ее номер или -1
               GRBALPHABET t,           // первый символ цепочки
               Rule::Chain& pchain,     // возвращаемая цепочка
               short j);                // номер цепочки
     };

     struct Greibach // грамматика Грейбах
     {
          short size;                   // количество правил
          GRBALPHABET startN;      // стартовый символ
          GRBALPHABET stbottomT;   // дно стека
          Rule* rules;             // множество правил

          Greibach()
          {
               size = 0;
               startN = 0;
               stbottomT = 0;
               rules = 0;
          };
          Greibach(GRBALPHABET pstartN, // стартовый символ
               GRBALPHABET pstbottomT,       // дно стека
               short psize,                  // количество правил
               Rule r, ...);                 // правила
```

```
          short getRule(                  // получить правило,
возвращается номер правила или -1
                GRBALPHABET pnn,          // левый символ правила
                Rule& prule);             // возвращаемое правило
грамматики

          Rule getRule(short n);          // получить правило по
номеру
     };

     Greibach getGreibach();        // получить грамматику
}
```

Листинг 4. Разбор исходного кода синтаксическим анализатором

```
Шаг :Правило              Входная лента                    Стек
0    :S->tdi(F){NrU;}S    tdi(tai){?(l>i):{p(l);}!{     S$
1    : SAVESTATE:          1
1                                                                :
tdi(tai){?(l>i):{p(l);}!{     tdi(F){NrU;}S$
2      :                              di(tai){?(l>i):{p(l);}!{p
di(F){NrU;}S$
3                                                                :
i(tai){?(l>i):{p(l);}!{p(     i(F){NrU;}S$
4      :                              (tai){?(l>i):{p(l);}!{p(i
(F){NrU;}S$
5      :                              tai){?(l>i):{p(l);}!{p(i)
F){NrU;}S$
6       :F->tP                        tai){?(l>i):{p(l);}!{p(i)
F){NrU;}S$
7    : SAVESTATE:          2
7      :                              tai){?(l>i):{p(l);}!{p(i)
tP){NrU;}S$
8      :                              ai){?(l>i):{p(l);}!{p(i);
P){NrU;}S$
9       :P->ai                        ai){?(l>i):{p(l);}!{p(i);
P){NrU;}S$
10   : SAVESTATE:          3
10     :                              ai){?(l>i):{p(l);}!{p(i);
ai){NrU;}S$
11     :                              i){?(l>i):{p(l);}!{p(i);}
i){NrU;}S$
12   :                   ){?(l>i):{p(l);}!{p(i);}r     ){NrU;}S$
13   :                   {?(l>i):{p(l);}!{p(i);}rl     {NrU;}S$
14   :                   ?(l>i):{p(l);}!{p(i);}rl;     NrU;}S$
15   :N->?KJN            ?(l>i):{p(l);}!{p(i);}rl;     NrU;}S$
16   : SAVESTATE:         4
16   :                    ?(l>i):{p(l);}!{p(i);}rl;      ?KJNrU;}S
$
```

```
17  :                      (l>i):{p(l);}!{p(i);}rl;}     KJNrU;}S$
18  :K->(U<U)              (l>i):{p(l);}!{p(i);}rl;}     KJNrU;}S$
19  : SAVESTATE:            5
19   :                                  (l>i):{p(l);}!{p(i);}rl;}
(U<U)JNrU;}S$
20   :                                  l>i):{p(l);}!{p(i);}rl;}m
U<U)JNrU;}S$
21   :U->l                              l>i):{p(l);}!{p(i);}rl;}m
U<U)JNrU;}S$
22  : SAVESTATE:            6
22   :                                  l>i):{p(l);}!{p(i);}rl;}m
l<U)JNrU;}S$
23  :                      >i):{p(l);}!{p(i);}rl;}m{     <U)JNrU;}
S$
24  : TNS_NS_NORULECHAIN/NS_NORULE
24  : RESTATE
24   :                                  l>i):{p(l);}!{p(i);}rl;}m
U<U)JNrU;}S$
25  : TNS_NS_NORULECHAIN/NS_NORULE
25  : RESTATE
25  :                      (l>i):{p(l);}!{p(i);}rl;}     KJNrU;}S$
26  :K->(U>U)              (l>i):{p(l);}!{p(i);}rl;}     KJNrU;}S$
27  : SAVESTATE:            5
27   :                                  (l>i):{p(l);}!{p(i);}rl;}
(U>U)JNrU;}S$
28   :                                  l>i):{p(l);}!{p(i);}rl;}m
U>U)JNrU;}S$
29   :U->l                              l>i):{p(l);}!{p(i);}rl;}m
U>U)JNrU;}S$
30  : SAVESTATE:            6
30   :                                  l>i):{p(l);}!{p(i);}rl;}m
l>U)JNrU;}S$
31  :                      >i):{p(l);}!{p(i);}rl;}m{     >U)JNrU;}
S$
32   :                                  i):{p(l);}!{p(i);}rl;}m{t
U)JNrU;}S$
33   :U->i                              i):{p(l);}!{p(i);}rl;}m{t
U)JNrU;}S$
34  : SAVESTATE:            7
34   :                                  i):{p(l);}!{p(i);}rl;}m{t
i)JNrU;}S$
35  :                     ):{p(l);}!{p(i);}rl;}m{tv     )JNrU;}S$
36  :                     :{p(l);}!{p(i);}rl;}m{tvi     JNrU;}S$
37  :J->:{N}              :{p(l);}!{p(i);}rl;}m{tvi     JNrU;}S$
38  : SAVESTATE:            8
38  :                      :{p(l);}!{p(i);}rl;}m{tvi     :{N}NrU;}
S$
39   :                                  {p(l);}!{p(i);}rl;}m{tvi=
{N}NrU;}S$
40  :                     p(l);}!{p(i);}rl;}m{tvi=i     N}NrU;}S$
41  :N->p(W);N            p(l);}!{p(i);}rl;}m{tvi=i     N}NrU;}S$
42  : SAVESTATE:            9
```

```
42    :                                    p(l);}!{p(i);}rl;}m{tvi=i
p(W);N}NrU;}S$
43                                                                 :
(l);}!{p(i);}rl;}m{tvi=i(     (W);N}NrU;}S$
44    :                                    l);}!{p(i);}rl;}m{tvi=i(l
W);N}NrU;}S$
45    :W->l                                l);}!{p(i);}rl;}m{tvi=i(l
W);N}NrU;}S$
46  : SAVESTATE:          10
46    :                                    l);}!{p(i);}rl;}m{tvi=i(l
l);N}NrU;}S$
47  :                     );}!{p(i);}rl;}m{tvi=i(l)     );N}NrU;}
S$
48  :                     ;}!{p(i);}rl;}m{tvi=i(l);     ;N}NrU;}S
$
49  :                    }!{p(i);}rl;}m{tvi=i(l);}     N}NrU;}S$
50  :N->                 }!{p(i);}rl;}m{tvi=i(l);}     N}NrU;}S$
51  : SAVESTATE:          11
51  :                    }!{p(i);}rl;}m{tvi=i(l);}     }NrU;}S$
52  :                    !{p(i);}rl;}m{tvi=i(l);}$     NrU;}S$
53  :N->                 !{p(i);}rl;}m{tvi=i(l);}$     NrU;}S$
54  : SAVESTATE:          12
54  :                    !{p(i);}rl;}m{tvi=i(l);}$     rU;}S$
55  : TNS_NS_NORULECHAIN/NS_NORULE
55  : RESTATE
55  :                    !{p(i);}rl;}m{tvi=i(l);}$     NrU;}S$
56  : TNS_NS_NORULECHAIN/NS_NORULE
56  : RESTATE
56  :                    }!{p(i);}rl;}m{tvi=i(l);}     N}NrU;}S$
57  : TNS_NS_NORULECHAIN/NS_NORULE
57  : RESTATE
57    :                                    l);}!{p(i);}rl;}m{tvi=i(l
W);N}NrU;}S$
58    :W->l,W                            l);}!{p(i);}rl;}m{tvi=i(l
W);N}NrU;}S$
59  : SAVESTATE:          10
59    :                                    l);}!{p(i);}rl;}m{tvi=i(l
l,W);N}NrU;}S$
60  :                     );}!{p(i);}rl;}m{tvi=i(l)     ,W);N}NrU
;}S$
61  : TNS_NS_NORULECHAIN/NS_NORULE
61  : RESTATE
61    :                                    l);}!{p(i);}rl;}m{tvi=i(l
W);N}NrU;}S$
62  : TNS_NS_NORULECHAIN/NS_NORULE
62  : RESTATE
62  :                    p(l);}!{p(i);}rl;}m{tvi=i     N}NrU;}S$
63  :N->                 p(l);}!{p(i);}rl;}m{tvi=i     N}NrU;}S$
64  : SAVESTATE:          9
64  :                    p(l);}!{p(i);}rl;}m{tvi=i     }NrU;}S$
65  : TNS_NS_NORULECHAIN/NS_NORULE
65  : RESTATE
```

```
65  :                     p(l);}!{p(i);}rl;}m{tvi=i     N}NrU;}S$
66  : TNS_NS_NORULECHAIN/NS_NORULE
66  : RESTATE
66  :                    :{p(l);}!{p(i);}rl;}m{tvi     JNrU;}S$
67  :J->:{N}!{N}         :{p(l);}!{p(i);}rl;}m{tvi     JNrU;}S$
68  : SAVESTATE:          8
68  :                     :{p(l);}!{p(i);}rl;}m{tvi     :{N}!{N}N
rU;}S$
69   :                                {p(l);}!{p(i);}rl;}m{tvi=
{N}!{N}NrU;}S$
70   :                                p(l);}!{p(i);}rl;}m{tvi=i
N}!{N}NrU;}S$
71    :N->p(W);N                      p(l);}!{p(i);}rl;}m{tvi=i
N}!{N}NrU;}S$
72  : SAVESTATE:          9
72   :                                p(l);}!{p(i);}rl;}m{tvi=i
p(W);N}!{N}NrU;}S$
73                                                            :
(l);}!{p(i);}rl;}m{tvi=i(     (W);N}!{N}NrU;}S$
74   :                                l);}!{p(i);}rl;}m{tvi=i(l
W);N}!{N}NrU;}S$
75   :W->l                           l);}!{p(i);}rl;}m{tvi=i(l
W);N}!{N}NrU;}S$
76  : SAVESTATE:          10
76   :                                l);}!{p(i);}rl;}m{tvi=i(l
l);N}!{N}NrU;}S$
77  :                     );}!{p(i);}rl;}m{tvi=i(l)     );N}!{N}N
rU;}S$
78  :                     ;}!{p(i);}rl;}m{tvi=i(l);     ;N}!{N}Nr
U;}S$
79   :                                }!{p(i);}rl;}m{tvi=i(l);}
N}!{N}NrU;}S$
80   :N->                            }!{p(i);}rl;}m{tvi=i(l);}
N}!{N}NrU;}S$
81  : SAVESTATE:          11
81  :                     }!{p(i);}rl;}m{tvi=i(l);}     }!{N}NrU;
}S$
82  :                     !{p(i);}rl;}m{tvi=i(l);}$     !{N}NrU;}
S$
83                                                            :
{p(i);}rl;}m{tvi=i(l);}$      {N}NrU;}S$
84  :                    p(i);}rl;}m{tvi=i(l);}$       N}NrU;}S$
85  :N->p(W);N           p(i);}rl;}m{tvi=i(l);}$       N}NrU;}S$
86  : SAVESTATE:          12
86                                                            :
p(i);}rl;}m{tvi=i(l);}$       p(W);N}NrU;}S$
87                                                            :
(i);}rl;}m{tvi=i(l);}$        (W);N}NrU;}S$
88                                                            :
i);}rl;}m{tvi=i(l);}$         W);N}NrU;}S$
89                                                         :W->i
i);}rl;}m{tvi=i(l);}$         W);N}NrU;}S$
```

```
90  : SAVESTATE:          13
90                                                                  :
i);}rl;}m{tvi=i(l);}$          i);N}NrU;}S$
91  :                     );}rl;}m{tvi=i(l);}$          );N}NrU;}
S$
92  :                     ;}rl;}m{tvi=i(l);}$           ;N}NrU;}S
$
93  :                    }rl;}m{tvi=i(l);}$            N}NrU;}S$
94  :N->                 }rl;}m{tvi=i(l);}$            N}NrU;}S$
95  : SAVESTATE:          14
95  :                    }rl;}m{tvi=i(l);}$            }NrU;}S$
96  :                    rl;}m{tvi=i(l);}$             NrU;}S$
97  :N->                 rl;}m{tvi=i(l);}$             NrU;}S$
98  : SAVESTATE:          15
98  :                    rl;}m{tvi=i(l);}$             rU;}S$
99  :                    l;}m{tvi=i(l);}$              U;}S$
100 :U->l                l;}m{tvi=i(l);}$              U;}S$
101 : SAVESTATE:          16
101 :                    l;}m{tvi=i(l);}$              l;}S$
102 :                    ;}m{tvi=i(l);}$               ;}S$
103 :                    }m{tvi=i(l);}$                }S$
104 :                    m{tvi=i(l);}$                 S$
105 :S->m{N}S            m{tvi=i(l);}$                 S$
106 : SAVESTATE:          17
106 :                    m{tvi=i(l);}$                 m{N}S$
107 :                    {tvi=i(l);}$                  {N}S$
108 :                    tvi=i(l);}$                   N}S$
109 :N->tY;N             tvi=i(l);}$                   N}S$
110 : SAVESTATE:          18
110 :                    tvi=i(l);}$                   tY;N}S$
111 :                    vi=i(l);}$                    Y;N}S$
112 :Y->vi               vi=i(l);}$                    Y;N}S$
113 : SAVESTATE:          19
113 :                    vi=i(l);}$                    vi;N}S$
114 :                    i=i(l);}$                     i;N}S$
115 :                    =i(l);}$                      ;N}S$
116 : TNS_NS_NORULECHAIN/NS_NORULE
116 : RESTATE
116 :                    vi=i(l);}$                    Y;N}S$
117 :Y->vi=U             vi=i(l);}$                    Y;N}S$
118 : SAVESTATE:          19
118 :                    vi=i(l);}$                    vi=U;N}S$
119 :                    i=i(l);}$                     i=U;N}S$
120 :                    =i(l);}$                      =U;N}S$
121 :                    i(l);}$                       U;N}S$
122 :U->i                i(l);}$                       U;N}S$
123 : SAVESTATE:          20
123 :                    i(l);}$                       i;N}S$
124 :                    (l);}$                        ;N}S$
125 : TNS_NS_NORULECHAIN/NS_NORULE
125 : RESTATE
125 :                    i(l);}$                       U;N}S$
```

```
126 : TNS_NS_NORULECHAIN/NS_NORULE
126 : RESTATE
126 :                    vi=i(l);}$                      Y;N}S$
127 :Y->vi=i(W)          vi=i(l);}$                      Y;N}S$
128 : SAVESTATE:          19
128                                                                :
vi=i(l);}$                    vi=i(W);N}S$
129                                                                :
i=i(l);}$                     i=i(W);N}S$
130                                                                :
=i(l);}$                      =i(W);N}S$
131 :                    i(l);}$                         i(W);N}S$
132 :                    (l);}$                          (W);N}S$
133 :                    l);}$                           W);N}S$
134 :W->l                l);}$                           W);N}S$
135 : SAVESTATE:          20
135 :                    l);}$                           l);N}S$
136 :                    );}$                            );N}S$
137 :                    ;}$                             ;N}S$
138 :                    }$                              N}S$
139 :N->                 }$                              N}S$
140 : SAVESTATE:          21
140 :                    }$                              }S$
141 :                    $                               S$
142 :S->                 $                               S$
143 : SAVESTATE:          22
143 :                    $                               $
144 :
145 : LENTA_END
146 : ------>LENTA_END
```

Приложение Г

Листинг 1. Программная реализация механизма преобразования

```
bool polishNotation(int lextable_pos, LT::LexTable& lextable,
IT::IdTable& idtable)
	{
		stack<LT::Entry*> stk;	//создаем стек для хранения
временных операций
		queue<LT::Entry*> result;
		bool function = false;
		int quantityParm = 0;
		int i = ++lextable_pos;

		for (; lextable.table[i]->lexema != LEX_SEMICOLON &&
(lextable.table[i]->lexema != LEX_RIGHTHESIS || !stk.empty()); i++)
		{
			switch (lextable.table[i]->lexema)
			{
			case LEX_ID:			//операнды
			case LEX_LITERAL:
				if (idtable.table[lextable.table[i]->idxTI]-
>idtype == IT::IDTYPE::F)
				{
					quantityParm = 0;
					function = true;
					result.push(lextable.table[i]);
					break;
				}
				if (function && !quantityParm)
					quantityParm++;

				result.push(lextable.table[i]);
				break;

			case LEX_OPERATOR:
				if (stk.empty() || stk.top()->lexema ==
LEX_LEFTHESIS)
					stk.push(lextable.table[i]);
				else
				{
					int prioritet = priority(lextable.table[i]-
>sign);
					if (priority(stk.top()->sign) >= prioritet)
					{
						result.push(stk.top());

						stk.pop();
					}
					stk.push(lextable.table[i]);
				}
				break;

			case LEX_LEFTHESIS:
				stk.push(lextable.table[i]);
				break;
```

```
                case LEX_RIGHTHESIS:
                      while (stk.top()->lexema != LEX_LEFTHESIS)
                      {
                            result.push(stk.top());

                            stk.pop();
                      }
                      stk.pop();

                      if (function)
                      {
                            result.push(new LT::Entry('@'));
                            result.push(new LT::Entry('0' +
quantityParm));
                            function = false;
                      }
                      break;

                case LEX_COMMA:
                      if (function)
                            quantityParm++;
                      while (stk.top()->lexema != LEX_LEFTHESIS)
                      {
                            result.push(stk.top());

                            stk.pop();
                      }
                      break;
                case LEX_MORE:
                case LEX_LESS:
                      result.push(lextable.table[i]);
                      break;
                }

          }
          while (!stk.empty())
          {
                result.push(stk.top());

                stk.pop();
          }
          for (int j = lextable_pos; j < i; j++)
          {
                if (!result.empty())
                {
                      lextable.table[j] = result.front();
                      lextable.table[j]->sn = lextable.table[j - 1]-
>sn;
                      lextable.table[j]->tn = lextable.table[j - 1]->tn
+ 1;
                      result.pop();
                }
                else
                {
                      lextable.table[j] = new LT::Entry('@',
lextable.table[j]->sn = lextable.table[j - 1]->sn, lextable.table[j]-
```

```
>tn = lextable.table[j - 1]->tn + 1);
                }
            }

            return true;
            }
```

Приложение Д

```
.586                                        ; система команд (процессор Pentium)
.model flat, stdcall                    ; модель памяти, соглашение о вызовах
includelib kernel32.lib
includelib libucrt.lib
includelib StdLib.lib

ExitProcess PROTO: dword        ; прототип функции для завершения процесса Windows

EXTRN strlens: proc
EXTRN print_int: proc
EXTRN print_str: proc
EXTRN atois: proc
EXTRN adate: proc
EXTRN atime: proc

.stack 4096

.const                                      ; сегмент констант – литералы
      L0 sdword 5
      L1 sdword 0
      L2 sdword 4
.data                                       ; сегмент данных – переменные и
параметры
      callf_main sdword 0
.code                                       ; сегмент кода

;----------- func ------------
func PROC,   ccc_func : sdword
; --- сохранить регистры ---
push ebx
push edx
; ----------------------
mov edx, L0
cmp edx, ccc_func

jg true0
jl false0
true0:
push L0
call print_int

jmp next0

false0:
push ccc_func
```

```
call print_int

next0:

; --- восстановить регистры ---
pop edx
pop ebx
; -------------------------
mov eax, L1
ret
func ENDP
;------------------------------


;----------- MAIN ------------
main PROC

push L2
call func
push eax
pop callf_main

push 0
call ExitProcess
main ENDP
end main
```