МИНИСТЕРСТВО ОБРАЗОВАНИЯ РЕСПУБЛИКИ БЕЛАРУСЬ

Учреждение образования «БЕЛОРУССКИЙ ГОСУДАРСТВЕННЫЙ ТЕХНОЛОГИЧЕСКИЙ УНИВЕРСИТЕТ»

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

ПОЯСНИТЕЛЬНАЯ ЗАПИСКА К КУРСОВОМУ ПРОЕКТУ НА ТЕМУ:

«Разработка компилятора SAA-2024»

Выполнил студент Соленок Анастасия Александровна
(Ф.И.О.)

Руководитель проекта преп.-ст. Некрасова Анастасия Павловна
(учен. степень, звание, должность, подпись, Ф.И.О.)

Заведующий кафедрой к.т.н., доц. Смелов Владимир Владиславович
(учен. степень, звание, должность, подпись, Ф.И.О.)

Консультанты преп.-ст. Некрасова Анастасия Павловна
(учен. степень, звание, должность, подпись, Ф.И.О.)

Нормоконтролер преп.-ст. Некрасова Анастасия Павловна
(учен. степень, звание, должность, подпись, Ф.И.О.)

Курсовой проект защищен с оценкой _____

Минск 2024

Содержание

Введение

В данном курсовом проекте мною поставлена задача создать удобный, понятный язык программирования SAA-2024, написанный на языке C++, и транслятор для данного языка, который будет транслировать SAA-2024 в язык ассемблера.

Транслятор – программа или средство, выполняющее преобразование программы, представленной на одном из языков программирования, в программу, написанную на другом языке.

В данном курсовом проекте поставлены следующие задачи:

– разработка спецификации SAA-2024;
– разработка структуры трансляции;
– разработка лексического анализатора;
– разработка синтаксического анализатора;
– разработка семантического анализатора;
– разбор арифметических выражений;
– тестирование транслятора.

1 Спецификация языка программирования

1.1 Характеристика языка программирования

Язык программирования SAA-2024 – это универсальный язык высокого уровня. Он является процедурным, компилируемым, не объектно-ориентированным. Язык строго типизируемый, что говорит о невозможности преобразования типов, транслируемым языком программирования.

1.2 Определение алфавит языка программирования

Совокупность символов, используемых в языке, называется алфавитом языка.

На этапе выполнения могут использоваться символы латинского алфавита, цифры десятичной системы счисления от 0 до 9, спецсимволы, а также непечатные символы пробела, табуляции и перевода строки. Русские символы разрешены только в строковых литералах.

1.3 Применяемые сепараторы

Символы-сепараторы служат в качестве разделителей операций языка. Сепараторы, используемые в языке программирования SAA-2024, приведены в таблице 1.1.

Таблица 1.1 – Сепараторы

Сепаратор	Название	Область применения
‘ ‘	Пробел	Допускается везде, кроме идентификаторов и ключевых слов
;	Точка с запятой	Разделение конструкций
{…}	Фигурные скобки	Заключение программного блока
[…]	Квадратные кавычки	Блок кода
(…)	Круглые скобки	Приоритет операций, параметры функции
“…”	Двойные кавычки	Строковый литерал
‘…’	Одинарные кавычки	Допускается везде, кроме идентификаторов и ключевых слов
=	Знак «равно»	Присваивание значения
,	Запятая	Разделение параметров
/ \ + - * :	Знаки «косая черта», «обратная косая черта», «плюс», «астерикс», «тильда»,	Выражения

Продолжение таблицы 1.1

Сепаратор	Название	Область применения
& ^ < >	«амперсанд», «циркумфлекс», знаки «больше» и «меньше»	Выражения в операторе цикла

1.4 Применяемые кодировки

Язык программирования SAA-2024 поддерживает работу с текстовыми данными, закодированными в формате Windows-1251. Это обеспечивает совместимость с устаревшими системами и позволяет эффективно обрабатывать кириллические символы.

1.5 Типы данных

В языке SAA-2024 реализованы четыре типа данных: целочисленный беззнаковый(uint), символьный(char), строковый(string). Описание типов данных, предусмотренных в данным языке представлено в таблице 1.2.

Таблица 1.2 – Типы данных языка SAA-2024

Тип данных	Описание типа данных
Целочисленный тип данных uint	Фундаментальный тип данных. Используется для работы с целочисленными положительными значениями. В памяти занимает 4 байта. При попытке инициализации значением больше максимального, инициализируется максимальным. При попытке инициализации значением меньше минимального, инициализируется значением, которое равно разнице между максимальным и исходным значением. Максимальное значение: 4294967295. Минимальное значение: 0. Инициализация по умолчанию: значение 0.
Строковый тип данных string	Фундаментальный тип данных. Используется для работы с набором символов, каждый символ в памяти занимает 1 байт. Инициализация по умолчанию: символ конца строки “\0”.
Символьный тип данных char	Фундаментальный тип данных. Используется для работы с символом, который в памяти занимает 1 байт. Инициализация по умолчанию: символ конца строки “\0”.

Пользовательские типы данных не поддерживаются.

1.6 Преобразование типов данных

Преобразование типов данных не поддерживается, т.е. язык является строготипизированным. Но в стандартной библиотеке есть функции преобразования типа uint в тип char и наоборот.

1.7 Идентификаторы

При создании идентификаторов можно использовать только латинские буквы, цифры и символ _. Длина идентификаторов ограничена 8 символами, а длина имен функций – 11 символами. Идентификаторы с большей длиной обрезаются. Имена идентификаторов не должны конфликтовать с именами стандартных функций, если только эти функции явно не импортированы через extern.

1.8 Литералы

С помощью литералов осуществляется инициализация переменных. В языке существует три типа литералов. Краткое описание литералов языка SAA-2024 представлено в таблице 1.3.

Таблица 1.3 – Описание литералов

Тип литерала	Регулярное выражение	Описание
Целочисленный литерал	[1-9]+[0-9]*	Целочисленные неотрицательные литералы, по умолчанию инициализируются 0. Литералы могут быть только rvalue.
Символьный литерал	[a-z|A-Z|A-Я|a-я|0-9|!-/]	Символ, заключённый в ‘’ (одинарные кавычки), по умолчанию инициализируются пустой строкой. Литералы могут быть только rvalue.
Строковый литерал	[a-z|A-Z|A-Я|a-я|0-9|!-/]*	Строковые литералы, максимальная длина строки 255 символов

1.9 Область видимости идентификаторов

Область видимости «сверху вниз» (по принципу C++). В языке SAA-2024 требуется обязательное объявление переменной перед её использованием. Все переменные должны находиться внутри программного блока языка. Имеется возможность объявления одинаковых переменных в разных блоках. Каждая переменная получает префикс – название функции, в которой она объявлена.

1.10 Инициализация данных

Инициализация переменных в языке SAA-2024 выполняется отдельно от их объявления. После объявления переменной ей можно присвоить значение с помощью оператора присваивания. Дополнительные правила и примеры инициализации переменных приведены в таблице 1.4.

Таблица 1.4 – Способы инициализации переменных

Конструкция	Примечание	Пример
declare <тип данных> <идентификатор>;	Автоматическая инициализация: переменные типа uint инициализируются нулём, переменные типа char – пустым символом.	declare uint sum; declare symbol chr;
<идентификатор> = <значение>;	Присваивание переменной значения.	sum = 9; chr = ‘D’;

1.11 Инструкции языка

Все возможные инструкции языка программирования SAA-2024 представлены в общем виде в таблице 1.5.

Таблица 1.5 – Инструкции языка программирования SAA-2024

Инструкция	Запись на языке SAA-2024
Объявление переменной	declare <тип данных> <идентификатор>;
Объявление функции	declare <тип данных> func <идентификатор> (<тип данных> <идентификатор>, …) {<блок кода>};
Присваивание	<идентификатор> = <значение>/<идентификатор>;
Объявление внешней функции	extern <тип данных> func <идентификатор> (<тип данных> <идентификатор>, …);

Блок инструкций	main
	{
	…
	}

Возврат из подпрограммы	return <идентификатор> / <литерал>;
Условная инструкция	while(<условие>)[<блок кода>];
Вывод данных	print <идентификатор> / <литерал>;
Однострочный комментарий до конца строки	#<любой текст>

1.12 Операции языка

Язык программирования SAA-2024 может выполнять операции, представленные в таблице 1.6. Операция сдвига учитывает только первый младший

бит оператора, т.к. сдвиг более чем на 255 любого числа кроме нуля вернет число большее, чем можно разместить в типе данных uint.

Таблица 1.6 – Операции языка программирования SAA-2024

Операция	Примечание	Типы данных	Пример
(	Приоритет операций	-	sum = (a + b) * c;
)			
+	Суммирование	(uint, uint) (char, char)	sum = a + b;
-	Вычитание	(uint, uint)	diff = a – b;
*	Умножение	(uint, uint)	mul = a*b;
:	Деление	(uint, uint)	div = a:b;
%	Остаток от деления	(uint, uint)	mod = a%b;
/	Сдвиг влево	(uint, uint)	pr = a / b;
\	Сдвиг вправо	(uint, uint)	pr = a \ b;
=	Присваивание	(uint, uint) (char, char) (string, string)	sum = 15; chr = ‘T’;
<,>	Знаки «больше», «меньше» для условной инструкции	(uint, uint) (char, char)	while(sum < diff) […];
&	Оператор эквивалентности	(uint, uint) (char, char)	while(sum & diff) […];
^	Оператор неравенства	(uint, uint) (char, char)	while(sum ^ diff) […];

Т.к. отрицательные числа не поддерживаются, если результат операции меньше нуля, он вычитается из максимального значения.

1.12 Выражения и их вычисления

Выражение — комбинация значений или переменных, констант, переменных, операций и функций, которая может быть интерпретирована в соответствии с правилами конкретного языка. Круглые скобки в выражении используются для изменения приоритета операций. Также не допускается запись двух подряд идущих арифметических операций. Выражение может содержать вызов функции, если эта функция уже содержится в стандартной библиотеке. Выражения вычисляются только после оператора присваивания. Результат вычисления выражения должен соответствовать типу данных переменной, которой присваивается значение.

Программные конструкции языка

Ключевые программные конструкции языка программирования SAA-2024 представлены в таблице 1.7.

Таблица 1.7 – Программные конструкции языка SAA-2024

Конструкция	Запись на языке SAA-2024

Главная функция (точка входа)	main
	{
	…
	}

Функция	declare <тип данных> func <идентификатор> (<тип> <идентификатор>, …) {… return <идентификатор> / <литерал>; };
Цикл	while(a^8)[…];
Условный оператор	if(5>4)[…];

Программные конструкции языка SAA-2024 представляют собой базовый функционал для выполнения различных операций, что делает возможным решать задачи различного уровня.

1.13 Область видимости

В языке SAA-2024 все переменные являются локальными, т.е. имеют функциональную область видимости. Они обязаны находится внутри программного блока функций (по принципу C++). Объявление глобальных переменных не предусмотрено. Объявление пользовательских областей видимости не предусмотрено.

1.14 Семантические проверки

Назначение семантического анализа – проверка смысловой правильности конструкций языка программирования. Таблица с перечнем семантических проверок, предусмотренных языком, приведена в таблице 1.8.

Таблица 1.8 – Семантические проверки

Номер	Правило
1	Идентификаторы не должны повторно объявляться в пределах одной функции.
2	Тип возвращаемого значения должен совпадать с типом функции при её объявлении или подключении
3	Тип данных передаваемых значений в функцию должен совпадать с типом параметров при её объявлении или подключении
4	В функцию должно быть передано то число параметров, сколько ожидается
5	Тип данных результата выражения должен совпадать с типом данных идентификатора, которому оно присваивается
6	Типы данных операндов выражения должны быть одинаковыми

Продолжение таблица 1.8

7	Тип данных string не может быть аргументом условной конструкции
8	Для типа char определены только операции + и -
9	Функции не должны подключаться дважды в пределах одной программы

Если семантическая проверка не проходит, то в лог журнал записывается соответствующая ошибка.

1.15 Распределение оперативной памяти на этапе выполнения

Все переменные размещаются в куче.

1.16 Стандартная библиотека и её состав

Чтобы воспользоваться функциями стандартной библиотеки, необходимо явно подключить нужную функцию с помощью ключевого слова extern. Это позволяет компилятору знать о наличии функции в другой части программы или в библиотеке. После подключения функции работа с ней осуществляется так же, как с пользовательскими функциями: вызов функции, передача аргументов и получение результата. Важно учитывать, что стандартная библиотека может содержать как стандартные, так и расширенные функции, которые могут требовать дополнительных настроек. Описание функций стандартной библиотеки приведено в таблице 1.9.

Таблица 1.9 – Состав стандартной библиотеки

Функция(C++)	Возвращаемое значение	Описание
int ord(char)	char	Возвращает код символа
char chr(int)	char	Возвращает символ с заданным кодом
int GetMonth(int number)	uint	Возвращает номер месяца
int GetDate(int number)	uint	Возвращает дату в формате ДДММГГГГ
int GetHours(int number)	uint	Возвращает час
int GetMinutes(int number)	uint	Возвращает минуты

Так же в библиотеке присутствуют приватные функции. Их описание представлено в таблице 1.10.

Таблица 1.10 – Приватные функции стандартной библиотеки

Функция(C++)	Возвращаемое значение	Описание
void outputuint (unsigned int a)	–	Выводит число на экран Вызывается оператором print
void outputchar (char a)	–	Выводит символ на экран Вызывается оператором print
void outputstring (void* in)	–	Выводит строку на экран Вызывается оператором print

Приватные функции не могут быть вызваны явно и не требуют предварительного пользовательского подключения. Они вызываются специальными операторами языка.

1.17 Ввод и вывод данных

Ввод данных не поддерживается языком программирования SAA-2024.

Для вывода данных в стандартный поток вывода предусмотрен оператор print, который базируется на приватных функциях стандартной библиотеки.

1.18 Точка входа

В языке SAA-2024 каждая программа обязана содержать главную функцию main, которая является точкой входа и с которой начинается последовательное выполнение программы.

1.19 Препроцессор

Препроцессор в языке программирования SAA-2024 не предусмотрен.

1.20 Соглашения о вызовах

В языке вызов функций происходит по соглашению о вызовах stdcall. Особенности stdcall:
– все параметры функции передаются через стек;
– память высвобождает вызываемый код;
– занесение в стек параметров идёт справа налево.

1.21 Объектный код

Трансляция кода SAA-2024 включает в себя генерацию ассемблерных инструкций, соответствующих логике программы, которые затем используются ассемблером для создания объектного кода.

1.22 Классификация сообщений транслятора

В случае возникновения ошибки в коде программы на языке SAA-2024 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
110-119	Ошибки открытия и чтения файлов
120-129	Ошибки лексического анализа
130-139	Ошибки таблиц лексем и таблиц идентификаторов
600-699	Ошибки синтаксического анализа
700-900	Ошибки семантического анализа

Контрольный пример

Код контрольного примера представлен в Приложении А.

2 Структура транслятора

2.1 Компоненты транслятора, принципы их взаимодействия

В языке SAA-2024 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 SAA-2024 приведена на рисунке 2.1.

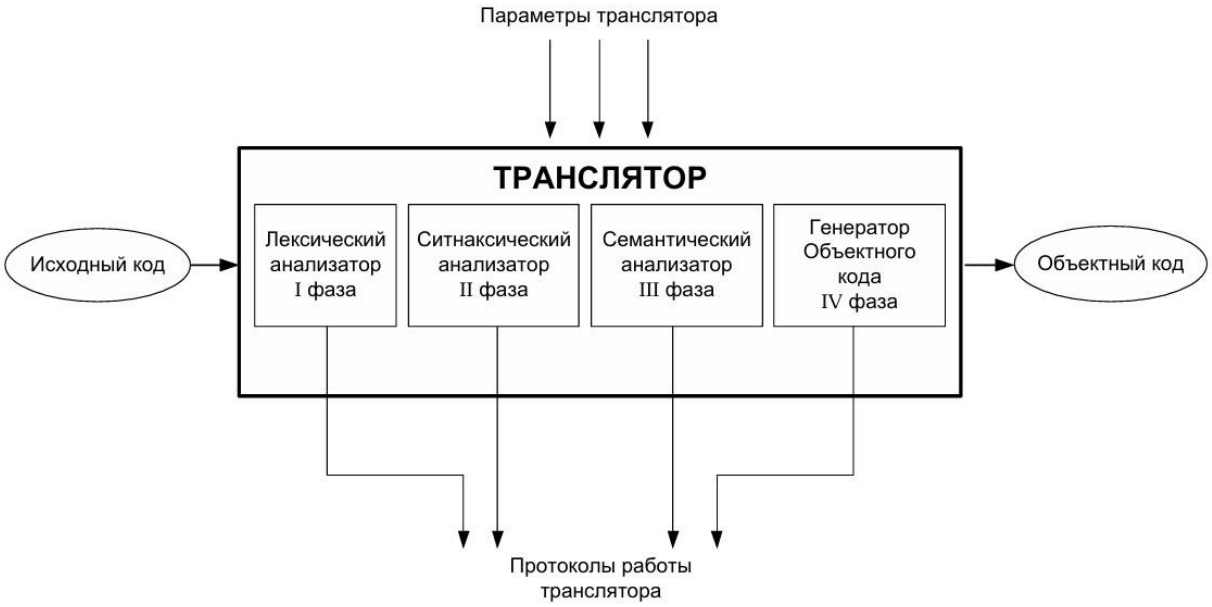


Рисунок 2.1 – Структура транслятора

Лексический анализ – первая фаза трансляции. Назначением лексического анализатора является нахождение ошибок лексики языка SAA-2024 и формирование таблицы лексем и таблицы идентификаторов.

Цели лексического анализатора:

− убрать все лишние пробелы;

− выполнить распознавание лексем;

− построить таблицу лексем и таблицу идентификаторов;

− при неуспешном распознавании или обнаружении некоторых ошибок во входном тексте выдать сообщение об ошибке.

Семантический анализ в свою очередь является проверкой исходной программы SAA-2024 на семантическую согласованность с определением языка, т.е. проверяет правильность текста исходной программы с точки зрения семантики.

Синтаксический анализ – это основная часть транслятора, предназначенная для распознавания синтаксических конструкций и формирования промежуточного кода SAA-2024. Для этого используются таблица лексем и идентификаторов. Синтаксический анализатор распознаёт синтаксические конструкции, выявляет синтаксические ошибки при их наличии и формирует дерево разбора

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 SAA-2024, прошедший успешно все предыдущие этапы, в код на языке Ассемблера.

2.2 Перечень входных параметров транслятора

Входные параметры представлены в таблице 2.1.

Таблица 2.1 – Входные параметры транслятора языка SAA-2024

Входной параметр	Описание	Значение по умолчанию
-in:<имя_файла>	Входной файл с любым расширением, в котором содержится исходный код на языке SAA-2024. Данный параметр должен быть указан обязательно. В случае если он не будет задан, то выполнение этапа трансляции не начнётся.	Не предусмотрено
-log:<имя_файла>	Файл содержит в себе краткую информацию об исходном коде на языке SAA-2024. В этот файл могут быть выведены таблицы идентификаторов, лексем, а также дерево разбора.	<имя_файла>.log
-out:<имя_файла>	В этот файл будет записан результат трансляции кода на язык assembler	<имя_файла>.asm
m	Вывод дерева разбора синтаксического анализатора.	–
l	Вывод таблицы лексем	–
i	Вывод таблицы идентификаторов	–

Таблицы лексем и дерево разбора синтаксического анализатора выводятся в лог журнал.

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 SAA-2024

Формируемый протокол	Описание протокола
Файл журнала, “*.log ”	Файл содержит в себе краткую информацию об исходном коде на языке SAA-2024. В этот файл выводится протокол работы анализаторов, а так же различные ошибки
“*.asm”	Содержит сгенерированный код на языке Ассемблера.

В log файл выводятся все ошибки, за исключением тех, что связаны с открытием файла log или считывания параметров.

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.

Функции лексического анализатора:

− Удаление «пустых» символов.

− Распознавание идентификаторов и ключевых слов.

− Распознавание литералов.

− Распознавание разделителей и знаков операций

Структура лексического анализатора представлена на рисунке 3.1.

Рисунок 3.1 – Структура лексического анализатора

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

3.2 Контроль входных символов

Исходный код на языке программирования SAA-2024 прежде чем транслироваться проверяется на допустимость символов. То есть изначально из

входного файла считывается по одному символу и проверяется является ли он разрешённым.

Таблица для контроля входных символов представлена в приложении Б.

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 – пробельный символ, C – символ одинарной кавычки, L – символ-разделитель, D – символ двойной кавычки, O – символ начала комментария, N – символ новой строки.

3.3 Удаление избыточных символов

Удаление избыточных символов не предусмотрено, так как после проверки на допустимость символов исходный код на языке программирования SAA-2024 разбивается на токены, которые записываются в очередь.

3.4 Перечень ключевых слов

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	Примечание	Цепочка	Лексема
Тип данных	Целочисленный беззнаковый тип данных	uint	i
	Строковый тип данных	string	i
	Символьный тип данных	char	i
Лексема	Объявление переменной	declare	v
	Подключение функции библиотеки	extern	e
	Оператор вывода	print	p
	Объявление функции	func	f
	Возврат значения из функции	return	r
	Инструкция цикла	while	u
	Инструкция Условного оператора	if	o
	Блок инструкции цикла	[	[
		]	]
	Блок функции	{	{
		}	}
	Оператор присваивания	=	v

Продолжение таблицы 3.1

Тип цепочки	Примечание	Цепочка	Лексема
	Изменение приоритетности в выражении и отделение параметров функций	(	(
		)	)
	Сепараторы	;	;
		,	,
	Условный оператор	<	b
		>	b
		&	b
		^	b
Оператор	Знаки арифметических операций	+	v
		-	v
		*	v
		/	v
		\	v
		:	v
		%	v
Идентификатор		[a-z|A-Z]+ [a-z|A-Z|0-9]*	i
Литерал	Целочисленный литерал	[1-9]+[0-9]*	l
	Символьный литерал	[a-z|A-Z|0-9]* кроме ‘	l
	Строковый литерал	[a-z|A-Z|0-9]* кроме ”	l
Точка входа		main	m

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 В.

Также в приложении В находятся конечные автоматы, соответствующие лексемам языка SAA-2024.

3.5 Основные структуры данных

Основные структуры таблиц лексем и идентификаторов данных языка SAA-2024, используемых для хранения, представлены в приложении В. В таблице лексем содержится лексема, её номер, полученный при разборе, номер строки в исходном коде, номер столбца в исходном коде, индекс таблицы

идентификаторов (если нет соответствующего идентификатора, то индекс равен -1), а также специальное поле, в котором хранится значение лексемы. В таблице идентификаторов содержится имя идентификатора, номер в таблице лексем, тип данных, тип идентификатора, его значение, а также бинарное поле для определения внешний ли идентификатор.

3.6 Структура и перечень сообщений лексического анализатора

Для обработки ошибок лексический анализатор использует таблицу с сообщениями. Структура сообщений содержит информацию о номере сообщения, номер строки и позицию, где было вызвано сообщение в исходном коде, информацию об ошибке. Перечень сообщений об ошибках лексического анализатора представлен на рисунке 3.2.

```
ERROR_ENTRY(120, "[LA]: Ошибка при разборе токена"),
ERROR_ENTRY(121, "[LA]: Используется необъявленный идентификатор"),
ERROR_ENTRY(122, "[LA]: Идентификатор не имеет типа"),
ERROR_ENTRY_NODEF(123),
ERROR_ENTRY(124, "[LA]: Отсутствует точка входа"),
ERROR_ENTRY(125, "[LA]: Обнаружена вторая точка входа"),
```

Рисунок 3.2 – Перечень ошибок лексического анализатора

При возникновении сообщения, лексический анализатор выбрасывает исключение – работа программы останавливается.

3.7 Структура и перечень сообщений лексического анализатора

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

3.8 Параметры лексического анализатора

Входные параметры используются для вывода результата работы лексического анализатора. Они передаются аргументами через командную строку и рассмотрены в таблице 2.1

3.9 Алгоритм лексического анализа

Алгоритм работы лексического анализа заключается в последовательном распознавании и разборе цепочек исходного кода и заполнение таблиц идентификаторов и лексем. Лексический анализатор производит распознаёт и разбирает цепочки исходного текста программы. Это основывается на работе

конечных автоматов, которую можно представить в виде графов. В случае, если подходящий автомат не был обнаружен, запоминается номер строки, в которой находился этот токен и выводится сообщение об ошибке. Если токен разобран, то дальнейшие действия, которые будут с ним производиться, будут зависеть от того, чем он является. 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.

В случае, если токен является идентификатором, перед его именем записывается название функции, в которой он объявлен и после этого он заносится в таблицу идентификаторов.

В случае, если токен является идентификатором функции, название функции в которой он объявлен не записывается.

В случае, если токен является литералом, то он заносится в таблицу идентификаторов в виде abi, где a – имя функции, где объявлен литерал, b – “$LEX”, c –количество определённых литералов+1.

Когда встречаем токен, являющийся ключевым словом, которое отвечает за тип данных или вид идентификатора, заносим лексему, соответствующую ему, в таблицу лексем и запоминаем тип данных или вид идентификатора, которому он соответствует.

В последствии, когда встречаем идентификатор, заносим его в таблицу идентификаторов с соответствующим ему типом данных и видом идентификатора, и именем вида “ab”, где a – имя функции, где объявлен идентификатор, b – имя идентификатора.

Пример. Регулярное выражение для ключевого слова declare: «declare».

Граф конечного автомата для этой лексемы представлен на рисунке 3.4. S0 – начальное состояние, S7 – конечное состояние автомата.

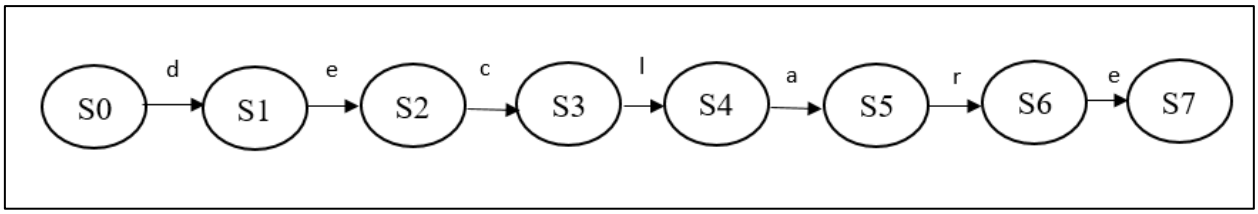


Рисунок 3.3 – Граф переходов для цепочки «declare»

3.10 Контрольный пример

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

4 Разработка синтаксического анализатора

4.1 Структура синтаксического анализатора

Синтаксический анализ – это фаза трансляции, выполняемая после лексического анализа и предназначенная для распознавания синтаксических конструкций SAA-2024. 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.

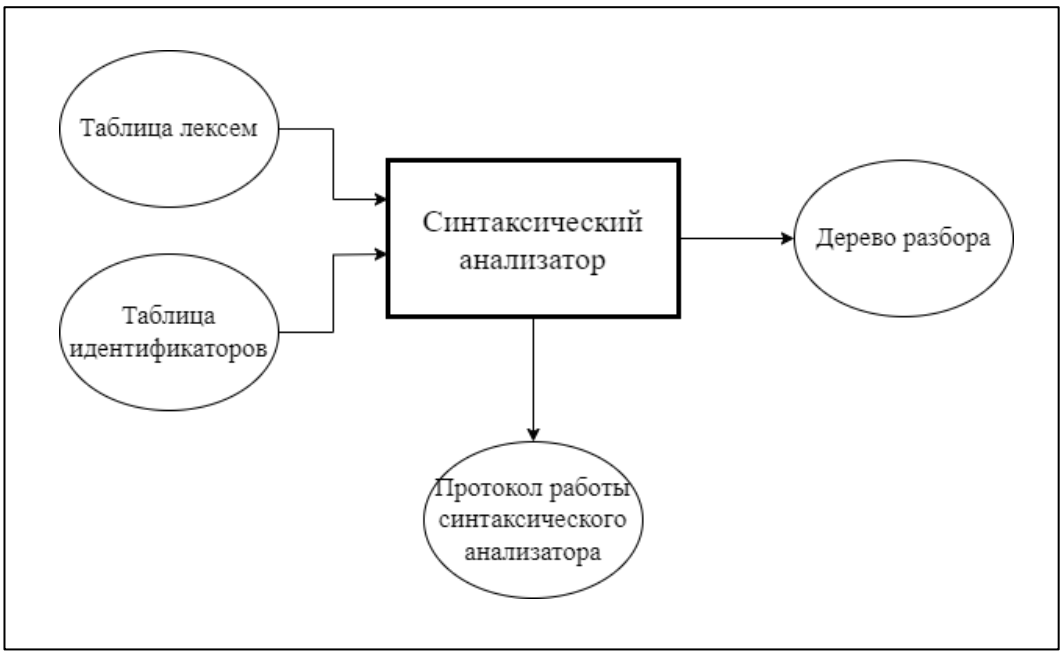


Рисунок 4.1 – Структура синтаксического анализатора SAA-2024

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 SAA-2024 используется контекстно-свободная грамматика $G = \langle T, N, P, S \rangle$ [2], 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 SAA-2024 представлена в приложении Г.

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 и описание нетерминальных символов SAA-2024

Нетерминал	Цепочки правил	Описание
S	m{NrE;}; m{rE;}; dtfi(F){NrE;};S dtfi(F){rE;};S dtfi(){NrE;};S dtfi(){rE;};S m{rE;};S	Порождает правила, описывающее общую структуру программы

N	dti;
	rE;
	ivE;
	etfi(F);
	o(B)[N]N
	o(B)[N]
	dti;N
	ivE;N
	etfi(F);N
	pi;
	pl;
	pi;N
	pl;N
	u(B)[N];
	u(B)[N];N
	etfi();N
	etfi();
		Порождает правила, описывающие конструкции языка

E	i l (E) i(W) iM lM	Порождает правила, описывающие выражения

Продолжение таблицы 4.1

Нетерминал	Цепочки правил	Описание
	(E) i(W) i() i(M) l(M)	
E	(E)M i(W)M i(W)	Порождает правила, описывающие выражения
F	ti ti,F	Порождает правила, описывающие параметры локальной функции при её объявлении
W	i l i,W l,W	Порождает правила, описывающие принимаемые параметры функции
B	ibi ibl lbi lbl	Порождает правила, описывающие условное выражение в операторе цикла
M	vE v(E) v(E)M vEM	Порождает правила, описывающие знаки арифметических операций

На основе этой грамматики синтаксический анализатор строит синтаксическое дерево, представляющее собой абстрактное представление структуры программы.

4.3 Построение конечного магазинного автомата

Конечный автомат с магазинной памятью представляет собой семерку $M = \langle Q, V, Z, \delta, q_0, z_0, F \rangle$[1], описание которой представлено в таблице 4.2. Структура данного автомата показана в приложении Г.

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

Продолжение таблицы 4.2– Описание компонентов магазинного автомата

Компонента	Определение	Описание
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 A)
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

Для вывода результата работы синтаксического анализатора нужно использовать флаг m.

4.4 Основные структуры данных

Основные структуры данных синтаксического анализатора включают в себя структуру магазинного конечного автомата и структуру грамматики Грейбах, описывающей правила языка SAA-2024. Данные структуры представлены в приложении Г.

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 и формируется дерево разбора. Иначе генерируется исключение.

4.6 Структура и перечень сообщений синтаксического анализатора

Индексы ошибок, обнаруживаемых синтаксическим анализатором, находятся в диапазоне 600-609. Перечень сообщений синтаксического анализатора представлен на рисунке 4.1.

```
ERROR_ENTRY(600, "[Syntaxis]: Неверная структура программы"),
ERROR_ENTRY(601, "[Syntaxis]: Ошибочный оператор"),
ERROR_ENTRY(602, "[Syntaxis]: Ошибка в выражении"),
ERROR_ENTRY(603, "[Syntaxis]: Ошибка в параметрах функции"),
ERROR_ENTRY(604, "[Syntaxis]: Ошибка в параметрах вызываемой функции"),
ERROR_ENTRY(605, "[Syntaxis]: Ошибка знака в выражении"),
ERROR_ENTRY(606, "[Syntaxis]: Ошибка синтаксического анализа"),
ERROR_ENTRY(607, "[Syntaxis]: Ошибка условной конструкции"),
ERROR_ENTRY_NODEF(608),
ERROR_ENTRY(609, "[Syntaxis]: Обнаружена синтаксическая ошибка (смотри журнал Log)"),
```

Рисунок 4.1 – Перечень сообщений синтаксического анализатора

Текст синтаксической ошибки содержит в себе префикс [Syntaxis].

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

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) В случае ошибки выводится соответствующее сообщение в журнал лога и компилятор прекращает работу.

4.9 Контрольный пример

Пример разбора синтаксическим анализатором исходного кода на языке SAA-2024 представлен в приложении Д. Дерево разбора исходного кода также представлено в приложении Д.

5 Разработка семантического анализатора

5.1 Структура семантического анализатора

Семантический анализатор получает на вход результаты работы лексического и синтаксического анализаторов, включая таблицы лексем и идентификаторов, а также дерево разбора. Он последовательно проверяет наличие ошибок. Некоторые проверки, такие как уникальность точки входа и предварительное объявление переменной, выполняются во время лексического анализа.

Семантический анализатор гарантирует корректность и согласованность семантической структуры программы на языке SAA-2024.

Структура семантического анализатора представлена на рисунке 5.1.

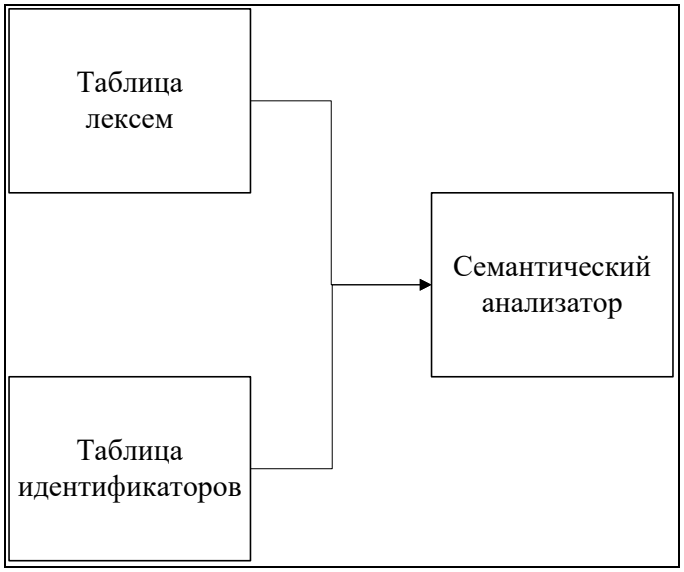


Рисунок 5.1 – Структура семантического анализатора

Функции семантического анализатора частично реализованы в лексическом анализаторе.

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

Все ошибка семантического анализатора имеют идентификатор свыше 700. Сообщения, формируемые семантическим анализатором, представлены на рисунке 5.2.

```
ERROR_ENTRY(700, "[Semantic]: Повторное объявление идентификатора"),
ERROR_ENTRY(701, "[Semantic]: Ошибка в возвращаемом значении"),
ERROR_ENTRY(702, "[Semantic]: Ошибка в передаваемых значениях в функции: количество параметров не совпадает"),
ERROR_ENTRY(703, "[Semantic]: Ошибка в передаваемых значениях в функции: типы параметров не совпадают"),
ERROR_ENTRY(704, "[Semantic]: Нарушены типы данных в выражении"),
ERROR_ENTRY(705, "[Semantic]: Ошибка экспорта: в библиотеке нет такой функции"),
ERROR_ENTRY(706, "[Semantic]: Ошибка экспорта: неверные параметры"),
ERROR_ENTRY(707, "[Semantic]: Ошибка экспорта: ошибочный тип возвращаемого значения"),
ERROR_ENTRY(708, "[Semantic]: Ошибочный оператор: строки можно только складывать"),
ERROR_ENTRY(709, "[Semantic]: Ошибочные параметры условной конструкции: строки не могут быть параметрами условной конструкции"),
ERROR_ENTRY(710, "[Semantic]: Ошибочный опреатор: для типа char разрешены только операции + и -"),
```

Рисунок 5.2 – Перечень сообщений семантического анализатора

Текст семантической ошибки содержит в себе префикс [Semantic].

5.4 Принцип обработки ошибок

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

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 семантическую проверки одновременно.

6 Вычисление выражений

6.1 Выражения, допускаемые языком

В языке SAA-2024 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 SAA-2024

Операция	Значение приоритета
(	1
)	1
+	2
-	2
*	3
:	3
%	3
/	3
\	3

Некоторые из операций таблицы 6.1 используются для типов, отличных от целочисленных.

6.2 Польская запись

Выражения в языке SAA-2024 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:

– читаем очередной символ;

– если он является идентификатором или литералом, то добавляем его к выходной строке;

– если символ является символом функции, то помещаем его в стек;

– если символ является открывающей скобкой, то она помещается в стек;

– исходная строка просматривается слева направо;

– если символ является закрывающей скобкой, то выталкиваем из стека в выходную строку все символы пока не встретим открывающую

скобку. При этом обе скобки удаляются и не попадают в выходную строку;

– как только входная лента закончится все символы из стека выталкиваются в строку;

– в случае если встречаются операции, то выталкиваем из стека в выходную строку все операции, которые имеют выше приоритетность чем последняя операция;

– также, если идентификатор является именем функции, то он заменяется на спецсимвол «@».

Таблица 6.2 – Пример преобразования выражения в обратную польскую запись

Исходная строка	Результирующая строка	Стек
x+y*5/(z-2)		
+y*5/(z-2)	x	
y*5/(z-2)	x	+
*5/(z-2)	xy	+
5/(z-2)	xy	+*
/(z-2)	xy5	+*
(z-2)	xy5*	+/
z-2)	xy5*	+/(
-2)	xy5*z	+/(
2)	xy5*z	+/(-
)	xy5*z2	+/(-
	xy5*z2-	+/
	xy5*z2-/	+
	xy5*z2-/+	

Как результат успешного разбора, мы получаем пустой стек и заполненную результирующую строку.

6.3 Программная реализация обработки выражений

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

6.4 Контрольный пример

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

7 Генерация кода

7.1 Структура генератора кода

Генерация объектного кода — это процесс, при котором компилятор преобразует внутреннее представление исходной программы SAA-2024 в последовательность символов выходного языка. На этом этапе компилятор использует таблицы лексем и идентификаторов, созданные на предыдущих стадиях компиляции. Эти таблицы содержат информацию о переменных, функциях, типах данных и других элементах программы, которые необходимы для построения ассемблерного кода. Сгенерированный ассемблерный код затем может быть использован для создания объектных файлов, которые в дальнейшем связываются для получения исполнимой программы.

Рисунок 7.1 – Структура генератора кода

7.2 Представление типов данных в оперативной памяти

Элементы таблицы идентификаторов размещаются в различных сегментах ассемблерного языка, таких как .data и .const, в зависимости от их типа и назначения. Сегмент .data используется для хранения переменных, значения которых могут изменяться во время выполнения программы, в то время как сегмент .const предназначен для хранения констант, которые остаются неизменными. Важно отметить, что правильно распределенные элементы иденти-

фикаторов помогают оптимизировать использование памяти и ускорить выполнение программы. Соответствия между типами данных идентификаторов на языке SAA-2024 и языке ассемблера, а также правила их размещения, приведены в таблице 7.1.

Таблица 7.1 – Соответствия типов идентификаторов языка SAA-2024 и языка Ассемблера

Тип идентификатора на языке SAA-2024	Тип идентификатора на языке ассемблера	Пояснение
char	BYTE	Хранит символьный тип данных.
string	DWORD	Хранит указатель на начало строки.
uint	DWORD	Хранит целочисленный тип данных без знака.
Лексема	BYTE DWORD DWORD	Литералы: символьные, целочисленные, строковые

Идентификаторы языка SAA-2024 размещены в сегменте данных (.data). Литералы – в сегменте констант (.const).

7.3 Статическая библиотека

Статическая библиотека реализована на языке программирования C++. Её код находится в проекте SAA-2024LIB, в свойствах которого был выбран пункт «статическая библиотека .lib». Подключение библиотеки в языке ассемблера происходит с помощью директивы includelib на этапе генерации кода. Далее объявляются имена функций из библиотеки. Вышеописанное проиллюстрировано в листинге 7.1.

```
void Head(std::ofstream* stream, LEX::LEX t) {

          *stream << ".586\n";
          *stream << "\t.model flat, stdcall\n";
          *stream << "\tincludelib libucrt.lib\n";
          *stream << "\tincludelib kernel32.lib\n";
          *stream << *stream << "\tincludelib C:/Us-
ers/Hasee/Desktop/кп/SAA-2024-master/SAA-2024-master/SAA-
2024/Debug/SAA-2024LIB.lib\n";

          *stream << "\tExitProcess PROTO :DWORD\n\n";
          for (int i = 0; i < t.idtable.size; i++)
          {
               if (t.idtable.table[i].idtype == IT::F)
               {    //Если библиотечная
```

```
                    if (t.idtable.table[i].isExternal == true)
                    {
                         *stream  <<  "\n\t"  <<  t.idtable.ta-
ble[i].id << " PROTO";
                         int pos = 1;
                         bool commaFlag = false;
                         while (true)
                         {
                              if (t.lextable.table[t.idtable.ta-
ble[i].idxfirstLE + pos].lexema == LEX_ID
                                   &&
                                   t.idtable.ta-
ble[t.lextable.table[t.idtable.table[i].idxfirstLE            +
pos].idxTI].idtype == IT::P)
                                   {
                                   if (commaFlag)
                                   {
                                        *stream << ',';
                                   }
```

Листинг 7.1 – Фрагмент функции генерации кода

7.4 Алгоритм работы генератора кода

Алгоритм генерации кода выглядит следующим образом:

1) Генерирует заголовочную информацию (Лист. 7.2): модель памяти, подключение библиотек, прототипы внешних функций, размер стека.

```
.586
     .model flat, stdcall
     includelib libucrt.lib
     includelib kernel32.lib
     includelib  C:/Users/Hasee/Desktop/кп/SAA-2024-master/SAA-
2024-master/SAA-2024/Debug/SAA-2024LIB.lib
     ExitProcess PROTO :DWORD
     GetHours PROTO :DWORD
     GetMonth PROTO :DWORD
     GetMinutes PROTO :DWORD
     GetDate PROTO :DWORD
outputuint PROTO :DWORD
outputchar PROTO :BYTE
outputstring PROTO :DWORD
```

Листинг 7.2 –Заголовочная информация

1) Проходит полностью таблицу идентификаторов и заполняет поле .const литералами (Лист. 7.3).

```
.const
     divideOnZeroExeption BYTE "Попытка деления на ноль.", 0
     ;STRING, для вывода ошибки при делении на ноль
     FindFactor$LEX1 DWORD 1 ;INT
     main$LEX4 BYTE "Stringing:", 0  ;STRING
     main$LEX5 BYTE "okay", 0  ;STRING
     main$LEX6 BYTE 'q' ;CHR
     main$LEX7 BYTE "Symbol", 0  ;STRING
     main$LEX8 BYTE "Factorial of number 5", 0  ;STRING
     main$LEX9 DWORD 5 ;INT
     main$LEX10 BYTE "Number to be circilar shifted:", 0
;STRING
     main$LEX11 DWORD 32 ;INT
     main$LEX12 DWORD 3 ;INT
     main$LEX14 BYTE "32<<3:", 0  ;STRING
     main$LEX15 BYTE "32>>1:", 0  ;STRING
     main$LEX16 DWORD 6 ;INT
     main$LEX18 BYTE "If constringuction works", 0  ;STRING
     main$LEX21 BYTE "If constringuction not works", 0  ;STRING
     main$LEX26 BYTE "Hours:", 0  ;STRING
     main$LEX27 BYTE "Minutes:", 0  ;STRING
     main$LEX28 BYTE "Date:", 0  ;STRING
     main$LEX29 BYTE "Month:", 0  ;STRING
     main$LEX30 DWORD 0 ;INT
```

Листинг 7.3 – Пример заполнения поля .const

2) Проходим таблицу идентификаторов и объявляем переменные в поле .data. (Лист. 7.4).

```
.data
     FindFactoranswer DWORD 0 ;INT
     mainstring1 DWORD 0 ;STRING
     mainsymb BYTE 0 ;CHR
     mainnumber DWORD 0 ;INT
     maindemo DWORD 0 ;INT
     maindemo1 DWORD 0 ;INT
     maindemo2 DWORD 0 ;INT
     mainnumber1 DWORD 0 ;INT
     mainnumber2 DWORD 0 ;INT
     mainnumber3 DWORD 0 ;INT
     mainnumber4 DWORD 0 ;INT
```

Листинг 7.4 – Пример заполнения поля .data

3) Генерируем сегмент данных .code (Лист. 7.5). Сперва проходим по таблице идентификаторов и ищем функции. Объявляем их и генерируем код, содержащийся в функциях. Так же перед именем функции дописываем знак «$», чтобы исключить совпадение имени функции с ключевым словом ассемблера. При генерации кода, при встрече оператора присваивания, описываем вычисление выражения. Описание алгоритма преобразования выражений представлено в пункте 7.3.

```
.code
     $FindFactor PROC uses ebx ecx edi esi ,FindFactora: DWORD
     ; Stringing #3 :ivl
     push FindFactor$LEX1
     pop FindFactoranswer

     While17Start:
     mov eax, FindFactora
     mov ebx, FindFactor$LEX1
     cmp eax, ebx
     jl While17End

     ; Stringing #6 :iviiv
     push FindFactoranswer
     push FindFactora
     pop ebx
     pop eax
     mul ebx
     push eax
     pop FindFactoranswer

     ; Stringing #7 :ivilv
     push FindFactora
     push FindFactor$LEX1
     pop ebx
     pop eax
     sub eax, ebx
     push eax
     pop FindFactora
     jmp While17Start
     While17End:

     mov eax, FindFactoranswer
     ret
     $FindFactor ENDP
```

Листинг 7.5 – Пример заполнения поля .code

После генерации всех пользовательских функций, генерируется функция начала программы main в функции main по такому же принципу.

7.5 Контрольный пример

Генерируемый код записывается в файл заданный параметром “-out”. Сгенерированный код можно посмотреть в приложении Ж.

8 Тестирование транслятора

8.1 Тестирование фазы проверки на допустимость символов

В языке SAA-2024 не разрешается использовать запрещённые входным алфавитом символы где-либо кроме строковых или символьных переменных. Результат использования запрещённого символа показан в таблице 8.1.

Таблица 8.1 – Тестирование фазы проверки на допустимость символов

Исходный код	Диагностическое сообщение
main{в}	Ошибка 111: [IN]: Недопустимый символ в исходном файле (-in), строка 1, столбец 4

Эти ошибки служат важным инструментом для выявления и коррекции проблем в исходном коде ещё на ранних этапах компиляции.

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

main{
declare uint 1s;
}
	Ошибка 120: [LA]: Ошибка при разборе токена, строка 2, столбец 11

main{
s = 5;
}
	Ошибка 121: [LA]: Используется необъявленный идентификатор, строка 2, столбец 1

declare uint func a(){
declare q;
}
	Ошибка 122: [LA]: Идентификатор не имеет типа, строка 2, столбец 6

declare uint func a(){
declare uint q;
}
	Ошибка 124: [LA]: Отсутствует точка входа

main(){
declare char a;
}
main(){
declare char z;
}
	Ошибка 125: [LA]: Обнаружена вторая точка входа, строка 4, столбец 1

Ошибка лексического анализатора приводит к прекращению выполнения программы и записи соответствующей ошибки в лог журнал.

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

main{
declare char a
}
	Ошибка 609: Обнаружена синтаксическая ошибка(смотри журнал Log)

Ошибка синтаксического анализатора также приводит к прекращению выполнения программы и записи соответствующей ошибки в лог журнал.

8.4 Тестирование семантического анализатора

Итоги тестирования семантического анализатора приведены в таблице 8.4.

Таблица 8.4 – Тестирование семантического анализатора

Исходный код	Диагностическое сообщение

declare uint func f(string a){
return 5;
};
main{
declare uint a;
a = f(a);
return a;
};
	Ошибка 703: [Semantic]: Ошибка в передаваемых значениях в функции: типы параметров не совпадают, строка 6, столбец 7

main{
declare char a;
declare string a;
}
	Ошибка 700: [Semantic]: Повторное объявление идентификатора, строка 3, столбец 10

declare uint func f(uint q){
declare string a;
return a;
};
main{
declare uint a;
a = f(3);
return 0;
};
	Ошибка 701: [Semantic]: Ошибка в возвращаемом значении, строка 3, столбец 6

Продолжение таблицы 8.4

Исходный код	Диагностическое сообщение

declare uint func f(uint a, uint b){
return 5;
};
main{
declare uint a;
a = f(a);
return a;
};
	Ошибка 702: [Semantic]: Ошибка в передаваемых значениях в функции: количество параметров не совпадает, строка 6, столбец 8

Ошибка семантического анализатора также приводит к прекращению выполнения программы и записи соответствующей ошибки в лог журнал.

Заключение

В результате выполнения всех ранее изложенных этапов был успешно разработан и реализован рабочий транслятор языка программирования SAA-2024 на язык ассемблера. Язык SAA-2024 поддерживает три основных типа данных — беззнаковый целочисленный (uint), строковый (string) и символьный (char), что обеспечивает широкие возможности для создания различных типов данных и структур. Особое внимание уделено обработке арифметических операций, включая семь базовых операций с целочисленным типом, а также поддержке скобок, которые задают приоритет выполнения операций.

На этапе семантического анализа был реализован механизм проверки соответствия исходного кода 10 строгим правилам, что гарантирует корректность программ и предотвращает множество потенциальных ошибок. Важной частью стандартной библиотеки являются шесть публичных и три приватные функции, которые существенно расширяют функциональные возможности языка и позволяют эффективно решать разнообразные задачи.

Данный транслятор не только демонстрирует высокую степень автоматизации процессов компиляции, но и служит основой для дальнейших улучшений и развития языка SAA-2024. Это достижение подтверждает мощный потенциал как языка, так и разработанного транслятора, открывая новые горизонты для пользователей и разработчиков в области

Список использованных источников

1. Карпов Ю. Теория и технология программирования. Основы построения трансляторов, 2005. – 272с.

2. Введение в теорию трансляторов [Электронный ресурс]. – Режим доступа: http://bourabai.ru/tpoi/compilers.htm. – Дата доступа: 15.11.2022.

3. Википедия: Обратная польская запись [Электронный ресурс]. – Режим доступа: https://en.wikipedia.org/wiki/Reverse_Polish_notation. – Дата доступа: 20.11.2022.

4. Альфред Ахо, Рави Сети, Джеффри Ульман, Моника Лам "Компиляторы. Принципы, технологии, инструменты"

ПРИЛОЖЕНИЕ А

```
declare uint function FindFactorial(uint a){
        declare uint answer;
        answer = 1;
        while(a>1)
        [
        answer = answer*a;
        a = a - 1;
        ];
        return answer;
        };
main
{
        extern uint function GetHours(uint a);
        extern uint function GetMonth(uint a);
        extern uint function GetMinutes(uint a);
        extern uint function GetDate(uint a);

        declare char symb;
        symb = 'q';
        print "Symbol";
        print symb;

        print "Factorial of number 5";
        declare uint number;
        number = FindFactorial(5);
        print number;

        print "Number to be circilar shifted:";
        declare uint demo;
        demo = 32;
        print demo;
        declare uint demo1;
        demo1 = demo/3;
        declare uint demo2;
        demo2 = demo\1;
        print "32<<3:";
        print demo1;
        print "32>>1:";
        print demo2;

        if(6 > 5)
        [
                print "If construction works";
        ]
        if(5 > 6)
        [
                print "If construction not works";
        ]

        declare uint number1;
        declare uint number2;
        declare uint number3;
        declare uint number4;
        number1 = GetHours(1);
        number2 = GetMinutes(1);
        number3 = GetDate(1);
        number4 = GetMonth(1);

        print "Hours:";
        print number1;
        print "Minutes:";
        print number2;
        print "Date:";
        print number3;
        print "Month:";
        print number4;
        return 0;
}
```

Рисунок 1 – Контрольный пример

ПРИЛОЖЕНИЕ Б

```
#define IN_CODE_TABLE {\
IN::T,/**/\
IN::F,/*┌*/\
IN::F,/*┐*/\
IN::F,/*└*/\
IN::F,/*┘*/\
IN::F,/*│*/\
IN::F,/*─*/\
IN::F,/*●*/\
IN::F,/*◘*/\
IN::S,/*tabulation*/\
IN::N,/*New line*/\
IN::F,/*♂*/\
IN::F,/*♀*/\
IN::T,/*carrege return*/\
IN::F,/*♫*/\
IN::F,/*☼*/\
IN::F,/*┼*/\
IN::F,/*◄*/\
IN::F,/*↕*/\
IN::F,/*‼*/\
IN::F,/*¶*/\
IN::F,/*┴*/\
IN::F,/*┬*/\
IN::F,/*┤*/\
IN::F,/*↑*/\
IN::F,/*├*/\
IN::F,/*left-arrow*/\
IN::F,/*←*/\
IN::F,/*└*/\
IN::F,/*↔*/\
IN::F,/*▲*/\
IN::F,/*▼*/\
IN::S,/*Space*/\
IN::F,/*!*/\
IN::D,/*"*/\
IN::O,/*#*/\
IN::F,/*$*/\
IN::L,/*%*/\
IN::L,/*&*/\
IN::C,/*'*/\
IN::L,/*(*/\
IN::L,/*)*/\
IN::L,/***/\
IN::L,/*+*/\
IN::L,/*,*/\
IN::L,/*-*/\
IN::F,/*.*/\
IN::L,/*/*/\
IN::T,/*0*/\
IN::T,/*1*/\
IN::T,/*2*/\
IN::T,/*3*/\
IN::T,/*4*/\
IN::T,/*5*/\
IN::T,/*6*/\
IN::T,/*7*/\
IN::T,/*8*/\
IN::T,/*9*/\
IN::L,/*:*/\
IN::L,/*;*/\
IN::L,/*<*/\
IN::L,/*=*/\
IN::L,/*>*/\
IN::F,/*?*/\
IN::F,/*@*/\
IN::T,/*A*/\
IN::T,/*B*/\
IN::T,/*C*/\
IN::T,/*D*/\
IN::T,/*E*/\
IN::T,/*F*/\
IN::T,/*G*/\
IN::T,/*H*/\
IN::T,/*I*/\
IN::T,/*J*/\
IN::T,/*K*/\
IN::T,/*L*/\
IN::T,/*M*/\
IN::T,/*N*/\
IN::T,/*O*/\
IN::T,/*P*/\
IN::T,/*Q*/\
IN::T,/*R*/\
IN::T,/*S*/\
IN::T,/*T*/\
IN::T,/*U*/\
IN::T,/*V*/\
IN::T,/*W*/\
IN::T,/*X*/\
IN::T,/*Y*/\
IN::T,/*Z*/\
IN::L,/*[*/\
IN::L,/*\*/\
IN::L,/*]*/\
IN::L,/*^*/\
IN::T,/*_*/\
IN::F,/*`*/\
IN::T,/*a*/\
IN::T,/*b*/\
IN::T,/*c*/\
IN::T,/*d*/\
IN::T,/*e*/\
IN::T,/*f*/\
IN::T,/*g*/\
IN::T,/*h*/\
IN::T,/*i*/\
IN::T,/*j*/\
IN::T,/*k*/\
IN::T,/*l*/\
IN::T,/*m*/\
IN::T,/*n*/\
IN::T,/*o*/\
IN::T,/*p*/\
IN::T,/*q*/\
IN::T,/*r*/\
IN::T,/*s*/\
IN::T,/*t*/\
IN::T,/*u*/\
IN::T,/*v*/\
IN::T,/*w*/\
IN::T,/*x*/\
IN::T,/*y*/\
IN::T,/*z*/\
IN::L,/*{*/\
IN::F,/*|*/\
IN::L,/*}*/\
IN::F,/*~*/\
IN::F,/**/\
IN::F,/**/\
IN::F,/**/\
IN::F,/**/\
IN::F,/**/\
IN::F,/**/\
IN::F,/**/\
IN::F,/**/\
IN::F,/**/\
IN::F,/**/\
IN::F,/**/\
IN::F,/**/\
IN::F,/**/\
IN::F,/**/\
IN::F,/**/\
IN::F,/**/\
IN::F,/**/\
IN::F,/**/\
IN::F,/**/\
IN::F,/**/\
IN::F,/**/\
IN::F,/**/\
IN::F,/**/\
IN::F,/**/\
IN::F,/**/\
IN::F,/**/\
IN::F,/**/\
IN::F,/**/\
IN::F,/**/\
IN::F,/**/\
IN::F,/**/\
IN::F,/**/\
IN::F,/**/\
IN::F,/**/\
IN::F,/* */\
IN::F,/**/\
IN::F,/**/\
IN::F,/**/\
IN::F,/*¤*/\
IN::F,/**/\
```

ПРИЛОЖЕНИЕ В

--------Таблица лексем--------

Позиция	№ строки	№ столбца	Лексема	Индекс таблицы идентификаторов
0	1	1	d	-1
1	1	9	t	-1
2	1	14	f	-1
3	1	23	i	0
4	1	36	(	-1
5	1	37	t	-1
6	1	42	i	1
7	1	43	)	-1
8	1	44	{	-1
9	2	2	d	-1
10	2	10	t	-1
11	2	15	i	2
12	2	21	;	-1
13	3	2	i	2
14	3	9	v	-1
15	3	11	l	3
16	3	12	;	-1
17	4	2	u	-1
18	4	7	(	-1
19	4	8	i	1
20	4	9	b	-1
21	4	10	l	3
22	4	11	)	-1
23	5	2	[	-1
24	6	2	i	2
25	6	9	v	-1
26	6	11	i	2
27	6	17	v	-1
28	6	18	i	1
29	6	19	;	-1
30	7	2	i	1
31	7	4	v	-1
32	7	6	i	1
33	7	8	v	-1
34	7	10	l	3
35	7	11	;	-1
36	8	2	]	-1
37	8	3	;	-1
38	9	2	r	-1
39	9	9	i	2
40	9	15	;	-1
41	10	2	}	-1
42	10	3	;	-1
43	11	1	m	-1
44	12	1	{	-1
45	13	2	e	-1
46	13	9	t	-1
47	13	14	f	-1
48	13	23	i	4
49	13	31	(	-1
50	13	32	t	-1
51	13	37	i	5
52	13	38	)	-1
53	13	39	;	-1
54	14	2	e	-1
55	14	9	t	-1
56	14	14	f	-1
57	14	23	i	6
58	14	31	(	-1
59	14	32	t	-1
60	14	37	i	7
61	14	38	)	-1
62	14	39	;	-1
63	15	2	e	-1
64	15	9	t	-1
65	15	14	f	-1
66	15	23	i	8
67	15	33	(	-1
68	15	34	t	-1

Рисунок 1– Таблица лексем

69	15	39	i	9
70	15	40	)	-1
71	15	41	;	-1
72	16	2	e	-1
73	16	9	t	-1
74	16	14	f	-1
75	16	23	i	10
76	16	30	(	-1
77	16	31	t	-1
78	16	36	i	11
79	16	37	)	-1
80	16	38	;	-1
81	18	2	d	-1
82	18	10	t	-1
83	18	15	i	12
84	18	19	;	-1
85	19	2	i	12
86	19	7	v	-1
87	19	11	l	13
88	19	12	;	-1
89	20	2	p	-1
90	20	16	l	14
91	20	17	;	-1
92	21	2	p	-1
93	21	8	i	12
94	21	12	;	-1
95	23	2	p	-1
96	23	31	l	15
97	23	32	;	-1
98	24	2	d	-1
99	24	10	t	-1
100	24	15	i	16
101	24	21	;	-1
102	25	2	i	16
103	25	9	v	-1
104	25	11	i	0
105	25	24	(	-1
106	25	25	l	17
107	25	26	)	-1
108	25	27	;	-1
109	26	2	p	-1
110	26	8	i	16
111	26	14	;	-1
112	28	2	p	-1
113	28	40	l	18
114	28	41	;	-1
115	29	2	d	-1
116	29	10	t	-1
117	29	15	i	19
118	29	19	;	-1
119	30	2	i	19
120	30	7	v	-1
121	30	9	l	20
122	30	11	;	-1
123	31	2	p	-1
124	31	8	i	19
125	31	12	;	-1
126	32	2	d	-1
127	32	10	t	-1
128	32	15	i	21
129	32	20	;	-1
130	33	2	i	21
131	33	8	v	-1
132	33	10	i	19
133	33	14	v	-1
134	33	15	l	22
135	33	16	;	-1
136	34	2	d	-1
137	34	10	t	-1
138	34	15	i	23
139	34	20	;	-1

Рисунок 1 (продолжение) – Таблица лексем

140	35	2	i	23
141	35	8	v	-1
142	35	10	i	19
143	35	14	v	-1
144	35	15	l	3
145	35	16	;	-1
146	36	2	p	-1
147	36	16	l	24
148	36	17	;	-1
149	37	2	p	-1
150	37	8	i	21
151	37	13	;	-1
152	38	2	p	-1
153	38	16	l	25
154	38	17	;	-1
155	39	2	p	-1
156	39	8	i	23
157	39	13	;	-1
158	41	2	o	-1
159	41	4	(	-1
160	41	5	l	26
161	41	7	b	-1
162	41	9	l	17
163	41	10	)	-1
164	42	2	[	-1
165	43	3	p	-1
166	43	32	l	27
167	43	33	;	-1
168	44	2	]	-1
169	45	2	o	-1
170	45	4	(	-1
171	45	5	l	17
172	45	7	b	-1
173	45	9	l	26
174	45	10	)	-1
175	46	2	[	-1
176	47	3	p	-1
177	47	36	l	28
178	47	37	;	-1
179	48	2	]	-1
180	50	2	d	-1
181	50	10	t	-1
182	50	15	i	29
183	50	22	;	-1
184	51	2	d	-1
185	51	10	t	-1
186	51	15	i	30
187	51	22	;	-1
188	52	2	d	-1
189	52	10	t	-1
190	52	15	i	31
191	52	22	;	-1
192	53	2	d	-1
193	53	10	t	-1
194	53	15	i	32
195	53	22	;	-1
196	54	2	i	29
197	54	10	v	-1
198	54	12	i	4
199	54	20	(	-1
200	54	21	l	3
201	54	22	)	-1
202	54	23	;	-1
203	55	2	i	30
204	55	10	v	-1
205	55	12	i	8
206	55	22	(	-1
207	55	23	l	3
208	55	24	)	-1
209	55	25	;	-1
210	56	2	i	31

Рисунок 1 (продолжение) – Таблица лексем

210	56	2	i	31
211	56	10	v	-1
212	56	12	i	10
213	56	19	(	-1
214	56	20	l	3
215	56	21	)	-1
216	56	22	;	-1
217	57	2	i	32
218	57	10	v	-1
219	57	12	i	6
220	57	20	(	-1
221	57	21	l	3
222	57	22	)	-1
223	57	23	;	-1
224	59	2	p	-1
225	59	16	l	33
226	59	17	;	-1
227	60	2	p	-1
228	60	8	i	29
229	60	15	;	-1
230	61	2	p	-1
231	61	18	l	34
232	61	19	;	-1
233	62	2	p	-1
234	62	8	i	30
235	62	15	;	-1
236	63	2	p	-1
237	63	15	l	35
238	63	16	;	-1
239	64	2	p	-1
240	64	8	i	31
241	64	15	;	-1
242	65	2	p	-1
243	65	16	l	36
244	65	17	;	-1
245	66	2	p	-1
246	66	8	i	32
247	66	15	;	-1
248	67	2	r	-1
249	67	9	l	37
250	67	10	;	-1
251	68	1	}	-1

Рисунок 1 (конец) – Таблица лексем

--------Таблица идентификаторов--------

Позиция	Имя	Вид	Тип данных	Значение
0	FindFactor	Функция	Число	0
1	FindFactora	Параметр	Число	0
2	FindFactoranswer	Переменная	Число	0
3	FindFactor$LEX1	Литерал	Число	1
4	GetHours	Функция	Число	0
5	mainGetHoursa	Параметр	Число	0
6	GetMonth	Функция	Число	0
7	mainGetMontha	Параметр	Число	0
8	GetMinutes	Функция	Число	0
9	mainGetMinutesa	Параметр	Число	0
10	GetDate	Функция	Число	0
11	mainGetDatea	Параметр	Число	0
12	mainsymb	Переменная	Символ	
13	main$LEX4	Литерал	Символ	q
14	main$LEX5	Литерал	Строка	"Symbol"
15	main$LEX6	Литерал	Строка	"Factorial of number 5"
16	mainnumber	Переменная	Число	0
17	main$LEX7	Литерал	Число	5
18	main$LEX8	Литерал	Строка	"Number to be circilar shifted:"
19	maindemo	Переменная	Число	0
20	main$LEX9	Литерал	Число	32
21	maindemo1	Переменная	Число	0
22	main$LEX10	Литерал	Число	3
23	maindemo2	Переменная	Число	0
24	main$LEX12	Литерал	Строка	"32<<3:"
25	main$LEX13	Литерал	Строка	"32>>1:"
26	main$LEX14	Литерал	Число	6
27	main$LEX16	Литерал	Строка	"If construction works"
28	main$LEX19	Литерал	Строка	"If construction not works"
29	mainnumber1	Переменная	Число	0
30	mainnumber2	Переменная	Число	0
31	mainnumber3	Переменная	Число	0
32	mainnumber4	Переменная	Число	0
33	main$LEX24	Литерал	Строка	"Hours:"
34	main$LEX25	Литерал	Строка	"Minutes:"
35	main$LEX26	Литерал	Строка	"Date:"
36	main$LEX27	Литерал	Строка	"Month:"
37	main$LEX28	Литерал	Число	0

Рисунок 2 – Таблица идентификаторов

```
    FST l_uint(
         str,
         5, //количество состояний
         NODE(1, RELATION('u', 1)),
         NODE(1, RELATION('i', 2)),
         NODE(1, RELATION('n', 3)),
         NODE(1, RELATION('t', 4)),
         NODE()
    );
    FST l_until(
         str,
         6, //количество состояний
         NODE(1, RELATION('w', 1)),
         NODE(1, RELATION('h', 2)),
         NODE(1, RELATION('i', 3)),
         NODE(1, RELATION('l', 4)),
         NODE(1, RELATION('e', 5)),
         NODE()
    );
    FST l_if(
         str,
         3, //количество состояний
         NODE(1, RELATION('i', 1)),
         NODE(1, RELATION('f', 2)),
         NODE()
    );


    FST l_extern(
         str,
         7, //количество состояний
         NODE(1, RELATION('e', 1)),
         NODE(1, RELATION('x', 2)),
         NODE(1, RELATION('t', 3)),
         NODE(1, RELATION('e', 4)),
         NODE(1, RELATION('r', 5)),
         NODE(1, RELATION('n', 6)),
         NODE()
    );

    FST l_str(
         str,
         7, //количество состояний
         NODE(1, RELATION('s', 1)),
         NODE(1, RELATION('t', 2)),
         NODE(1, RELATION('r', 3)),
         NODE(1, RELATION('i', 4)),
         NODE(1, RELATION('n', 5)),
         NODE(1, RELATION('g', 6)),
         NODE()
    );

    FST l_numberLiteral(
```

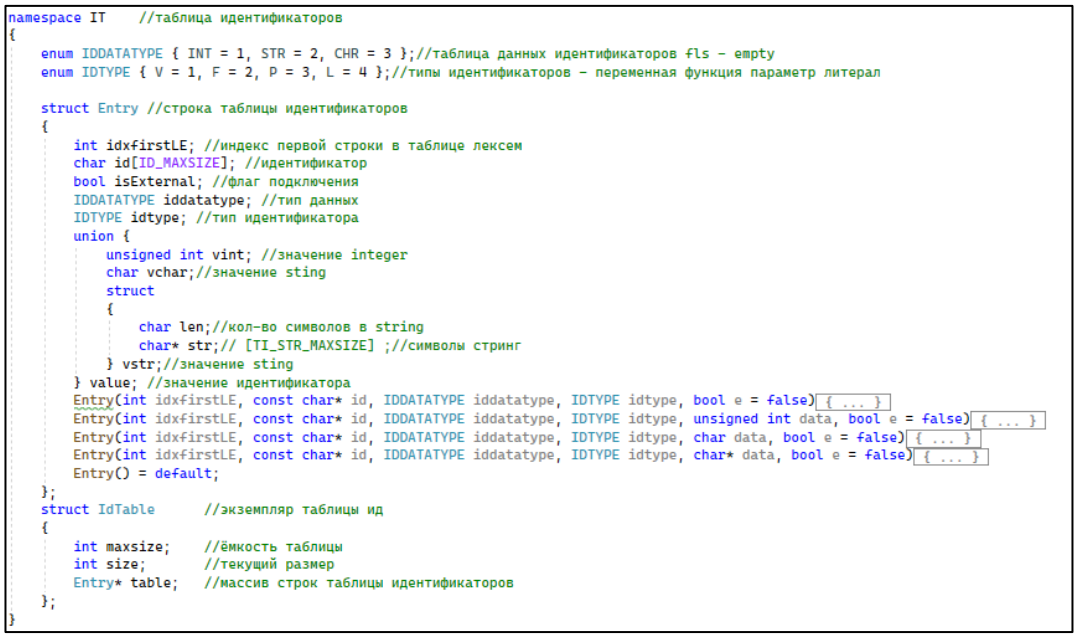
```
        str,
        2, //количество состояний
        NODE(20,

            RELATION('0', 0),
            RELATION('1', 0),
            RELATION('2', 0),
            RELATION('3', 0),
            RELATION('4', 0),
            RELATION('5', 0),
            RELATION('6', 0),
            RELATION('7', 0),
            RELATION('8', 0),
            RELATION('9', 0),

            RELATION('0', 1),
            RELATION('1', 1),
            RELATION('2', 1),
            RELATION('3', 1),
            RELATION('4', 1),
            RELATION('5', 1),
            RELATION('6', 1),
            RELATION('7', 1),
            RELATION('8', 1),
            RELATION('9', 1)
        ),
        NODE()
    );

    FST l_string(
        str,
        5, //количество состояний
        NODE(1, RELATION('c', 1)),
        NODE(1, RELATION('h', 2)),
        NODE(1, RELATION('a', 3)),
        NODE(1, RELATION('r', 4)),
        NODE()
    );

    FST l_function(
        str,
        9, //количество состояний
        NODE(1, RELATION('f', 1)),
        NODE(1, RELATION('u', 2)),
        NODE(1, RELATION('n', 3)),
        NODE(1, RELATION('c', 4)),
        NODE(1, RELATION('t', 5)),
        NODE(1, RELATION('i', 6)),
        NODE(1, RELATION('o', 7)),
        NODE(1, RELATION('n', 8)),
        NODE()
    );
    FST l_declare(
```

```
        str,
        8, //количество состояний
        NODE(1, RELATION('d', 1)),
        NODE(1, RELATION('e', 2)),
        NODE(1, RELATION('c', 3)),
        NODE(1, RELATION('l', 4)),
        NODE(1, RELATION('a', 5)),
        NODE(1, RELATION('r', 6)),
        NODE(1, RELATION('e', 7)),
        NODE()
    );
    FST l_return(
        str,
        7, //количество состояний
        NODE(1, RELATION('r', 1)),
        NODE(1, RELATION('e', 2)),
        NODE(1, RELATION('t', 3)),
        NODE(1, RELATION('u', 4)),
        NODE(1, RELATION('r', 5)),
        NODE(1, RELATION('n', 6)),
        NODE()
    );
    FST l_printi(
        str,
        6, //количество состояний
        NODE(1, RELATION('p', 1)),
        NODE(1, RELATION('r', 2)),
        NODE(1, RELATION('i', 3)),
        NODE(1, RELATION('n', 4)),
        NODE(1, RELATION('t', 5)),
        NODE()
    );

    FST l_prints(
        str,
        5, //количество состояний
        NODE(1, RELATION('s', 1)),
        NODE(1, RELATION('a', 2)),
        NODE(1, RELATION('y', 3)),
        NODE(1, RELATION('s', 4)),
        NODE()
    );

    FST l_main(
        str,
        5, //количество состояний
        NODE(1, RELATION('m', 1)),
        NODE(1, RELATION('a', 2)),
        NODE(1, RELATION('i', 3)),
        NODE(1, RELATION('n', 4)),
        NODE()
    );
    FST l_conditional(
```

```
        str,
        3, //количество состояний
        NODE(1, RELATION('i', 1)),
        NODE(1, RELATION('f', 2)),
        NODE()
    );

    FST l_semicolon(
        str,
        2, //количество состояний
        NODE(1, RELATION(';', 1)),
        NODE()
    );
    FST l_comma(
        str,
        2, //количество состояний
        NODE(1, RELATION(',', 1)),
        NODE()
    );
    FST l_braceleft(
        str,
        2, //количество состояний
        NODE(1, RELATION('{', 1)),
        NODE()
    );
    FST l_braceright(
        str,
        2, //количество состояний
        NODE(1, RELATION('}', 1)),
        NODE()
    );
    FST l_lefthesis(
        str,
        2, //количество состояний
        NODE(1, RELATION('(', 1)),
        NODE()
    );
    FST l_cycleStart(
        str,
        2, //количество состояний
        NODE(1, RELATION('[', 1)),
        NODE()
    );
    FST l_cycleEnd(
        str,
        2, //количество состояний
        NODE(1, RELATION(']', 1)),
        NODE()
    );
    FST l_conditionalStart(
        str,
        2, //количество состояний
        NODE(1, RELATION('[', 1)),
```

```
          NODE()
     );
     FST l_conditionalEnd(
          str,
          2, //количество состояний
          NODE(1, RELATION(']', 1)),
          NODE()
     );
     FST l_righthesis(
          str,
          2, //количество состояний
          NODE(1, RELATION(')', 1)),
          NODE()
     );

     FST l_verb(
          str,
          2, //количество состояний
          NODE(8, RELATION('+', 1), RELATION('-', 1), RELA-
TION('*', 1),
               RELATION('/', 1), RELATION(':', 1), RELA-
TION('\\', 1), RELATION('%', 1), RELATION('=', 1)),
          NODE()
     );
     FST l_boolVerb(
          str,
          2, //количество состояний
          NODE(4, RELATION('^', 1), RELATION('<', 1), RELA-
TION('>', 1), RELATION('&', 1)),
          NODE()
     );
```

Листинг 1 – Конечные автоматы



```
namespace IT    //таблица идентификаторов
{
    enum IDDATATYPE { INT = 1, STR = 2, CHR = 3 };//таблица данных идентификаторов fls - empty
    enum IDTYPE { V = 1, F = 2, P = 3, L = 4 };//типы идентификаторов - переменная функция параметр литерал

    struct Entry //строка таблицы идентификаторов
    {
        int idxfirstLE; //индекс первой строки в таблице лексем
        char id[ID_MAXSIZE]; //идентификатор
        bool isExternal; //флаг подключения
        IDDATATYPE iddatatype; //тип данных
        IDTYPE idtype; //тип идентификатора
        union {
            unsigned int vint; //значение integer
            char vchar;//значение sting
            struct
            {
                char len;//кол-во символов в string
                char* str;// [TI_STR_MAXSIZE] ;//символы стринг
            } vstr;//значение sting
        } value; //значение идентификатора
        Entry(int idxfirstLE, const char* id, IDDATATYPE iddatatype, IDTYPE idtype, bool e = false){ ... }
        Entry(int idxfirstLE, const char* id, IDDATATYPE iddatatype, IDTYPE idtype, unsigned int data, bool e = false){ ... }
        Entry(int idxfirstLE, const char* id, IDDATATYPE iddatatype, IDTYPE idtype, char data, bool e = false){ ... }
        Entry(int idxfirstLE, const char* id, IDDATATYPE iddatatype, IDTYPE idtype, char* data, bool e = false){ ... }
        Entry() = default;
    };
    struct IdTable      //экземпляр таблицы ид
    {
        int maxsize;    //ёмкость таблицы
        int size;       //текущий размер
        Entry* table;   //массив строк таблицы идентификаторов
    };
}
```

Рисунок 3 – Структура таблиц лексем и идентификаторов

ПРИЛОЖЕНИЕ Г

```
Rule(NS('S'), GRB_ERROR_SERIES + 0,
                6, //todo m{NrE;};     tfi(F){NrE;};S
m{NrE;};S           tfi(F){NrE;};
                Rule::Chain(7, TS('m'), TS('{'), NS('N'),
TS('r'), NS('E'), TS(';'), TS('}')),//+
                Rule::Chain(6, TS('m'), TS('{'), TS('r'),
NS('E'), TS(';'), TS('}')),//+
                Rule::Chain(15, TS('d'), TS('t'), TS('f'),
TS('i'), TS('('), NS('F'), TS(')'), TS('{'), NS('N'), TS('r'),
NS('E'), TS(';'), TS('}'), TS(';'), NS('S')),//+
                Rule::Chain(14, TS('d'), TS('t'), TS('f'),
TS('i'), TS('('), NS('F'), TS(')'), TS('{'), TS('r'), NS('E'),
TS(';'), TS('}'), TS(';'), NS('S')),
                Rule::Chain(14, TS('d'), TS('t'), TS('f'),
TS('i'), TS('('), TS(')'), TS('{'), NS('N'), TS('r'), NS('E'),
TS(';'), TS('}'), TS(';'), NS('S')),
                Rule::Chain(13, TS('d'), TS('t'), TS('f'),
TS('i'), TS('('), TS(')'), TS('{'), TS('r'), NS('E'), TS(';'),
TS('}'), TS(';'), NS('S'))
          ),
          Rule(NS('N'), GRB_ERROR_SERIES + 1,
                18, //todo dti;   rE;   i=E;   dtfi(F);   dti;N
rE;N   i=E;N   dtfi(F);N pl;N pi;N pl; pi; pi(W);
                Rule::Chain(4, TS('d'), TS('t'), TS('i'),
TS(';')),//+
                //Rule::Chain(5, TS('d'), TS('t'), TS('i'),
NS('M'), TS(';')),//
                Rule::Chain(5, TS('f'), TS('('), NS('F'),
TS(')'), TS(';')),//+
                Rule::Chain(6, TS('i'), TS('('), NS('F'),
TS(')'), TS(';'), NS('N')),//+
                Rule::Chain(8, TS('o'), TS('('), NS('B'),
TS(')'), TS('['), NS('N'), TS(']'), NS('N')),//+
                Rule::Chain(5, TS('o'), TS('('), NS('B'),
TS(')'), TS('['), NS('N'), TS(']')),//+
                Rule::Chain(5, TS('d'), TS('t'), TS('i'),
TS(';'), NS('N')), //+
                Rule::Chain(4, TS('i'), TS('v'), NS('E'),
TS(';')), //+
                Rule::Chain(5, TS('i'), TS('v'), NS('E'),
TS(';'), NS('N')),
                Rule::Chain(9, TS('e'), TS('t'), TS('f'),
TS('i'), TS('('), NS('F'), TS(')'), TS(';'), NS('N')), //+
                Rule::Chain(8, TS('e'), TS('t'), TS('f'),
TS('i'), TS('('), NS('F'), TS(')'), TS(';')),//+
                Rule::Chain(8, TS('e'), TS('t'), TS('f'),
TS('i'), TS('('), TS(')'), TS(';'), NS('N')),
                Rule::Chain(7, TS('e'), TS('t'), TS('f'),
TS('i'), TS('('), TS(')'), TS(';')),
                Rule::Chain(4, TS('p'), TS('i'), TS(';'),
NS('N')),//+
```

```
                Rule::Chain(3, TS('p'), TS('i'), TS(';')),//+
                Rule::Chain(4, TS('p'), TS('l'), TS(';'),
NS('N')),//+
                Rule::Chain(3, TS('p'), TS('l'), TS(';')),//+
                Rule::Chain(8, TS('u'), TS('('), NS('B'),
TS(')'), TS('['), NS('N'), TS(']'), TS(';')),
                Rule::Chain(9, TS('u'), TS('('), NS('B'),
TS(')'), TS('['), NS('N'), TS(']'), TS(';'), NS('N'))
           ),
           Rule(NS('E'), GRB_ERROR_SERIES + 2,
                9, //todo i   l   (E)   i(W)   iM   lM   (E)M
i(W)M
                Rule::Chain(1, TS('i')),
                Rule::Chain(1, TS('l')),
                Rule::Chain(3, TS('('), NS('E'), TS(')')),//+
                Rule::Chain(4, TS('i'), TS('('), NS('W'),
TS(')')),
                Rule::Chain(3, TS('i'), TS('('), TS(')')),
                Rule::Chain(2, TS('i'), NS('M')),//+
                Rule::Chain(2, TS('l'), NS('M')),//+
                Rule::Chain(4, TS('('), NS('E'), TS(')'),
NS('M')),//+
                Rule::Chain(5, TS('i'), TS('('), NS('W'),
TS(')'), NS('M'))
           ),
           Rule(NS('M'), GRB_ERROR_SERIES + 3,
                4, //todo vE    vEM   v(E) v(E)M
                Rule::Chain(2, TS('v'), NS('E')),//+
                Rule::Chain(4, TS('v'), TS('('), NS('E'),
TS(')')),
                Rule::Chain(5, TS('v'), TS('('), NS('E'),
TS(')'), NS('M')),
                Rule::Chain(3, TS('v'), NS('E'), NS('M'))
           ),
           Rule(NS('F'), GRB_ERROR_SERIES + 4,
                6, //todo ti   ti,F
                Rule::Chain(2, TS('t'), TS('i')),//+
                Rule::Chain(4, TS('t'), TS('i'), TS(','),
NS('F')),//+
                Rule::Chain(1, TS('i')),//+
                Rule::Chain(3, TS('i'), TS(','), NS('F')),
                Rule::Chain(1, TS('i')),//+
                Rule::Chain(3, TS('l'), TS(','), NS('F'))
           ),
           Rule(NS('W'), GRB_ERROR_SERIES + 5,
                4, //todo i   l   i,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(NS('B'), GRB_ERROR_SERIES + 6,
                4, //todo ibi   ibl   lbi   lbl
```

```
                Rule::Chain(3, TS('i'), TS('b'), TS('i')),
                Rule::Chain(3, TS('i'), TS('b'), TS('l')),
                Rule::Chain(3, TS('l'), TS('b'), TS('i')),
                Rule::Chain(3, TS('l'), TS('b'), TS('l'))
                                )
```

Листинг 1 – Грамматика языка SAA-2024

```
struct MfstDiagnosis    //диагностика
{
    short lenta_position;   //позиция на ленте
    RC_STEP rc_step;        //код завершения шага
    short nrule;            //номер правила
    short nrule_chain;      //номер цепочки правила
    MfstDiagnosis();
    MfstDiagnosis(          //диагностика
        short plenta_position,  //позиция на ленте
        RC_STEP prc_step,       //код завершения шага
        short pnrule,           //номер правила
        short pnrule_chain      //номер цепочки правила
    );
} diagnosis[MFST_DIAGN_NUMBER]; //последние самые глубокие сообщения??

GRBALPHABET* lenta;             //перекодированная лента из lex
short lenta_position;           //текущая позиция на ленте
short nrule;                    //номер текущего правила
short nrulechain;               //номер текущей цепочки, текущего правила
short lenta_size;               //размер ленты
GRB::Greibach greibach;         //грамматика Грейбах
LEX::LEX lex;                //результат работы лексического анализатора
MFSTSTACK st;                   //стек автомата
bool shallWrite;
use_container<std::stack<MfstState>> storestate;    //стек сохранения состояний
Mfst();
Mfst(
    LEX::LEX plex,              //результат работы лексического анализатора
    GRB::Greibach pgreibach,        //грамматика грейбах
    bool shouldWrite
);
char* getCSt(char* buf);        //получить содержимое стека
char* getCLenta(char* buf, short pos, short n = 25);        //лента: n символов с pos
char* getDiagnosis(short n, char* buf); //получить n-ю сумму диагностики или 0x00
bool savestate(const Log::LOG& log);                        //сохранить состояние автомата
bool reststate(const Log::LOG& log);                        //восстановитьсостояние автомата
bool push_chain(                    //поместить цепочку правила в стек
    GRB::Rule::Chain chain          //цепочка правила
);
RC_STEP step(const Log::LOG& log);                      //выполнить шаг автомата
bool start(const Log::LOG& log);                        //запустить автомат
bool savediagnosis(
    RC_STEP pprc_step               //код завершения шага
);
void printrules(const Log::LOG& lo);                    //вывести последовательность правил

struct Deducation                   //вывод
{
    short size;                     //кол-во шагов в выводе
    short* nrules;                  //номера правил в грамматике
    short* nrulechains;             //номера цепочек правил грамматики
    Deducation() { size = 0; nrules = 0; nrulechains = 0; };
} deducation;

bool savededucation();              //сохранить дерево вывода
```

Рисунок 1 – Структура конечного магазинного автомата

ПРИЛОЖЕНИЕ Д

```
Шаг :Правило              Входная лента                 Стек
0   :S->dtfi(F){NrE;};S   dtfi(ti){dti;ivl;u(ibl)[i     S$
1   : SAVESTATE:           1
1   :                     dtfi(ti){dti;ivl;u(ibl)[i     dtfi(F){NrE;};S$
2   :                     tfi(ti){dti;ivl;u(ibl)[iv     tfi(F){NrE;};S$
3   :                     fi(ti){dti;ivl;u(ibl)[ivi     fi(F){NrE;};S$
4   :                     i(ti){dti;ivl;u(ibl)[iviv     i(F){NrE;};S$
5   :                     (ti){dti;ivl;u(ibl)[ivivi     (F){NrE;};S$
6   :                     ti){dti;ivl;u(ibl)[ivivi;     F){NrE;};S$
7   :F->ti                ti){dti;ivl;u(ibl)[ivivi;     F){NrE;};S$
8   : SAVESTATE:           2
8   :                     ti){dti;ivl;u(ibl)[ivivi;     ti){NrE;};S$
9   :                     i){dti;ivl;u(ibl)[ivivi;i     i){NrE;};S$
10  :                     ){dti;ivl;u(ibl)[ivivi;iv     ){NrE;};S$
11  :                     {dti;ivl;u(ibl)[ivivi;ivi     {NrE;};S$
12  :                     dti;ivl;u(ibl)[ivivi;iviv     NrE;};S$
13  :N->dti;              dti;ivl;u(ibl)[ivivi;iviv     NrE;};S$
14  : SAVESTATE:           3
14  :                     dti;ivl;u(ibl)[ivivi;iviv     dti;rE;};S$
15  :                     ti;ivl;u(ibl)[ivivi;ivivl     ti;rE;};S$
16  :                     i;ivl;u(ibl)[ivivi;ivivl;     i;rE;};S$
17  :                     ;ivl;u(ibl)[ivivi;ivivl;]     ;rE;};S$
18  :                     ivl;u(ibl)[ivivi;ivivl;];     rE;};S$
19  : TS_NOK/NS_NORULECHAIN
```

Рисунок 1 – Начало синтаксического анализа

```
1190: SAVESTATE:           85
1190:                     pi;rl;}                       pi;NrE;}$
1191:                     i;rl;}                        i;NrE;}$
1192:                     ;rl;}                         ;NrE;}$
1193:                     rl;}                          NrE;}$
1194: TNS_NORULECHAIN/NS_NORULE
1194: RESSTATE
1194:                     pi;rl;}                       NrE;}$
1195:N->pi;               pi;rl;}                       NrE;}$
1196: SAVESTATE:           85
1196:                     pi;rl;}                       pi;rE;}$
1197:                     i;rl;}                        i;rE;}$
1198:                     ;rl;}                         ;rE;}$
1199:                     rl;}                          rE;}$
1200:                     l;}                           E;}$
1201:E->l                 l;}                           E;}$
1202: SAVESTATE:           86
1202:                     l;}                           l;}$
1203:                     ;}                            ;}$
1204:                     }                             }$
1205:                                                   $
1206: LENTA_END
1207: ------>LENTA_END
```

Рисунок 2 – Конец синтаксического анализа

```
0   : S->dtfi(F){NrE;};S
5   : F->ti
9   : N->dti;N
13  : N->ivE;N
15  : E->l
17  : N->u(B)[N];
19  : B->ibl
24  : N->ivE;N
26  : E->iM
27  : M->vE
28  : E->i
30  : N->ivE;
32  : E->iM
33  : M->vE
34  : E->l
39  : E->i
43  : S->m{NrE;}
45  : N->etfi(F);N
50  : F->ti
54  : N->etfi(F);N
59  : F->ti
63  : N->etfi(F);N
68  : F->ti
72  : N->etfi(F);N
77  : F->ti
81  : N->dti;N
85  : N->ivE;N
87  : E->l
89  : N->pl;N
92  : N->pi;N
95  : N->pl;N
98  : N->dti;N
102 : N->ivE;N
104 : E->i(W)
106 : W->l
109 : N->pi;N
112 : N->pl;N
115 : N->dti;N
119 : N->ivE;N
121 : E->l
123 : N->pi;N
126 : N->dti;N
130 : N->ivE;N
132 : E->iM
133 : M->vE
134 : E->l
136 : N->dti;N
140 : N->ivE;N
142 : E->iM
143 : M->vE
144 : E->l
146 : N->pl;N
149 : N->pi;N
152 : N->pl;N
155 : N->pi;N
158 : N->o(B)[N]N
160 : B->lbl
165 : N->pl;
169 : N->o(B)[N]N
171 : B->lbl
176 : N->pl;
180 : N->dti;N
184 : N->dti;N
188 : N->dti;N
192 : N->dti;N
196 : N->ivE;N
198 : E->i(W)
200 : W->l
203 : N->ivE;N
205 : E->i(W)
207 : W->l
210 : N->ivE;N
212 : E->i(W)
214 : W->l
217 : N->ivE;N
219 : E->i(W)
221 : W->l
224 : N->pl;N
227 : N->pi;N
230 : N->pl;N
233 : N->pi;N
236 : N->pl;N
239 : N->pi;N
242 : N->pl;N
245 : N->pi;
249 : E->l
```

Рисунок 3 – Пример разбора синтаксическим анализатором

ПРИЛОЖЕНИЕ Е

```
#include <stack>
#include <vector>
#include <iostream>
#include "PolishNotation.h"
#include "Error.h"

namespace PolishNotation {
    template <typename T>
    struct container : T
    {
        using T::T;
        using T::c;
    };

    std::string toString(int n) {
        char buf[40];
        sprintf_s(buf, "%d", n);
        return buf;
    }

    bool find_elem(std::stack<char> stack, size_t size, char elem) {
        for (size_t i = 0; i < size; i++)
            if (stack.top() == elem)
                return true;
            else
                stack.pop();
        return false;
    }
    int get_priority(char a)
    {
        switch (a)
        {
        case '(':
            return 0;
        case ')':
            return 0;
        case ',':
            return 1;
        case '-':
            return 2;
        case '+':
            return 2;
        case '*':
            return 3;
        case '%':
            return 3;
        case '/':
            return 3;
        case '\\':
            return 3;
        case ':':
            return 3;
        default: {
            return 0;
        }
        }
    }
```

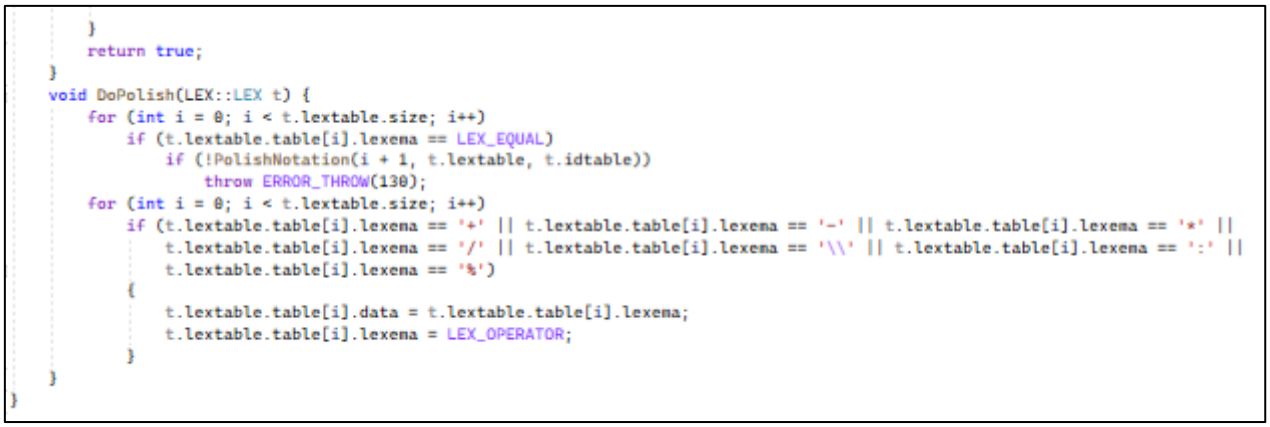
Рисунок 1 – Алгоритм преобразования выражения к польской записи

```
void fixIt(LT::LexTable& lextable, const std::string& str, size_t length, size_t pos, const std::vector<int>& ids) {
    for (size_t i = 0, q = 0; i < str.size(); i++) {
        lextable.table[pos + i].lexema = str[i];
        if (lextable.table[pos + i].lexema == LEX_ID || lextable.table[pos + i].lexema == LEX_LITERAL) {
            lextable.table[pos + i].idxTI = ids[q];
            q++;
        }
        else
            lextable.table[pos + i].idxTI = LT_TI_NULLIDX;
    }
    int temp = str.size() + pos;
    for (size_t i = 0; i < length - str.size(); i++) {
        lextable.table[temp + i].idxTI = LT_TI_NULLIDX;
        lextable.table[temp + i].lexema = '!';
        lextable.table[temp + i].sn = -1;
    }
}

bool PolishNotation(int lextable_pos, LT::LexTable& lextable, IT::IdTable& idtable)
{
    container<std::stack<char>> stack;
    std::string PolishString;
    std::vector<int> ids;
    int operators_count = 0, operands_count = 0, iterator = 0, right_counter = 0, left_counter = 0, params_counter = 0;

    for (int i = lextable_pos; i < lextable.size; i++, iterator++) {
        char lexem = lextable.table[i].lexema;
        char data = lextable.table[i].data;
        size_t stack_size = stack.size();
        if (idtable.table[lextable.table[i].idxTI].idtype == IT::IDTYPE::F) {
            stack.push('@');
            operands_count--;
        }
        switch (lexem) {
        case LEX_OPERATOR:
        {
            if (!stack.empty() && stack.top() != LEX_LEFTHESIS) {
                while (!stack.empty() && get_priority(data) <= get_priority(stack.top())) {
                    PolishString += stack.top();
                    stack.pop();
                }
            }
            stack.push(data);
            operators_count++;
            break;
        }
        case LEX_COMMA:
        {
            while (!stack.empty()) {
                if (stack.top() == LEX_LEFTHESIS)
                    break;
                PolishString += stack.top();
                stack.pop();
            }
            operands_count--;
            break;
```

Рисунок 1 (продолжение) – Алгоритм преобразования выражения к польской записи

```
}
case LEX_LEFTHESIS:
{
    left_counter++;
    stack.push(lexem);
    break;
}
case LEX_RIGHTHESIS:
{
    right_counter++;
    if (!find_elem(stack, stack_size, LEX_LEFTHESIS))
        return false;
    while (stack.top() != LEX_LEFTHESIS) {
        PolishString += stack.top();
        stack.pop();
    }
    stack.pop();
    if (!stack.empty() && stack.top() == '@') {
        PolishString += stack.top() + toString(params_counter - 1);
        params_counter = 0;
        stack.pop();
    }
    break;
}
case LEX_SEMICOLON:
{
    if (operators_count != 0 && operands_count != 0)
        if ((!stack.empty() && (stack.top() == LEX_RIGHTHESIS || stack.top() == LEX_LEFTHESIS))
            || right_counter != left_counter || operands_count - operators_count != 1)
            return false;
    while (!stack.empty()) {
        PolishString += stack.top();
        stack.pop();
    }
    fixIt(lextable, PolishString, iterator, lextable_pos, ids);
    return true;
    break;
}
case LEX_ID: {
    if (std::find(stack.c.begin(), stack.c.begin(), '@') != stack.c.end())
        params_counter++;
    PolishString += lexem;
    if (lextable.table[i].idxTI != LT_TI_NULLIDX)
        ids.push_back(lextable.table[i].idxTI);
    operands_count++;
    break;
}
case LEX_LITERAL: {
    if (std::find(stack.c.begin(), stack.c.begin(), '@') != stack.c.end())
        params_counter++;
    PolishString += lexem;
    if (lextable.table[i].idxTI != LT_TI_NULLIDX)
        ids.push_back(lextable.table[i].idxTI);
    operands_count++;
    break;
}
}
```

Рисунок 1 (продолжение) – Алгоритм преобразования выражения к польской записи



```
        }
        return true;
    }
    void DoPolish(LEX::LEX t) {
        for (int i = 0; i < t.lextable.size; i++)
            if (t.lextable.table[i].lexema == LEX_EQUAL)
                if (!PolishNotation(i + 1, t.lextable, t.idtable))
                    throw ERROR_THROW(130);
        for (int i = 0; i < t.lextable.size; i++)
            if (t.lextable.table[i].lexema == '+' || t.lextable.table[i].lexema == '-' || t.lextable.table[i].lexema == '*' ||
                t.lextable.table[i].lexema == '/' || t.lextable.table[i].lexema == '\\' || t.lextable.table[i].lexema == ':' ||
                t.lextable.table[i].lexema == '%')
            {
                t.lextable.table[i].data = t.lextable.table[i].lexema;
                t.lextable.table[i].lexema = LEX_OPERATOR;
            }
    }
}
```

Рисунок 1 (конец) – Алгоритм преобразования выражения к польской записи

ПРИЛОЖЕНИЕ Ж

```
.586
     .model flat, stdcall
     includelib libucrt.lib
     includelib kernel32.lib
     includelib C:/Users/Hasee/Desktop/кп/SAA-2024-master/SAA-
2024-master/SAA-2024/Debug/SAA-2024LIB.lib
     ExitProcess PROTO :DWORD


     GetHours PROTO :DWORD
     GetMonth PROTO :DWORD
     GetMinutes PROTO :DWORD
     GetDate PROTO :DWORD
outputuint PROTO :DWORD
outputchar PROTO :BYTE
outputstr PROTO :DWORD

.stack 4096
.const
divideOnZeroExeption BYTE "Попытка деления на ноль.", 0  ;STR,
для вывода ошибки при делении на ноль
     FindFactor$LEX1 DWORD 1 ;INT
     main$LEX4 BYTE 'q' ;CHR
     main$LEX5 BYTE "Symbol", 0  ;STR
     main$LEX6 BYTE "Factorial of number 5", 0  ;STR
     main$LEX7 DWORD 5 ;INT
     main$LEX8 BYTE "Number to be circilar shifted:", 0  ;STR
     main$LEX9 DWORD 32 ;INT
     main$LEX10 DWORD 3 ;INT
     main$LEX12 BYTE "32<<3:", 0  ;STR
     main$LEX13 BYTE "32>>1:", 0  ;STR
     main$LEX14 DWORD 6 ;INT
     main$LEX16 BYTE "If construction works", 0  ;STR
     main$LEX19 BYTE "If construction not works", 0  ;STR
     main$LEX24 BYTE "Hours:", 0  ;STR
     main$LEX25 BYTE "Minutes:", 0  ;STR
     main$LEX26 BYTE "Date:", 0  ;STR
     main$LEX27 BYTE "Month:", 0  ;STR
     main$LEX28 DWORD 0 ;INT
.data
     FindFactoranswer DWORD 0 ;INT
     mainsymb BYTE 0 ;CHR
     mainnumber DWORD 0 ;INT
     maindemo DWORD 0 ;INT
     maindemo1 DWORD 0 ;INT
     maindemo2 DWORD 0 ;INT
     mainnumber1 DWORD 0 ;INT
     mainnumber2 DWORD 0 ;INT
     mainnumber3 DWORD 0 ;INT
     mainnumber4 DWORD 0 ;INT
```

```
.code
$FindFactor PROC uses ebx ecx edi esi ,FindFactora: DWORD
; String #3 :ivl
push FindFactor$LEX1
pop FindFactoranswer

While17Start:
mov eax, FindFactora
mov ebx, FindFactor$LEX1
cmp eax, ebx
jl While17End

; String #6 :iviiv
push FindFactoranswer
push FindFactora
pop ebx
pop eax
mul ebx
push eax
pop FindFactoranswer

; String #7 :ivilv
push FindFactora
push FindFactor$LEX1
pop ebx
pop eax
sub eax, ebx
push eax
pop FindFactora
jmp While17Start
While17End:

mov eax, FindFactoranswer
ret
$FindFactor ENDP

main PROC

; String #19 :ivl
movzx eax, main$LEX4
push eax
pop eax
mov mainsymb, al

push offset main$LEX5
CALL outputstr
push eax
movzx eax, mainsymb
push eax
CALL outputchar
pop eax
```

```
push offset main$LEX6
CALL outputstr

; String #25 :ivil@1
invoke $FindFactor, main$LEX7
push eax ;результат функции
pop mainnumber

push mainnumber
CALL outputuint

push offset main$LEX8
CALL outputstr

; String #30 :ivl
push main$LEX9
pop maindemo

push maindemo
CALL outputuint

; String #33 :ivilv
push maindemo
push main$LEX10
pop ebx
pop eax
push ecx ; сохраняем данные регистра ecx
mov ecx, ebx
SHL eax, cl
pop ecx
push eax
pop maindemo1

; String #35 :ivilv
push maindemo
push FindFactor$LEX1
pop ebx
pop eax
push ecx ; сохраняем данные регистра ecx
mov ecx, ebx
SHR eax, cl
pop ecx
push eax
pop maindemo2

push offset main$LEX12
CALL outputstr

push maindemo1
CALL outputuint

push offset main$LEX13
CALL outputstr
```

```
push maindemo2
CALL outputuint

If158Start:
mov eax, main$LEX14
mov ebx, main$LEX7
cmp eax, ebx
jl If158End

push offset main$LEX16
CALL outputstr
If158End:

If169Start:
mov eax, main$LEX7
mov ebx, main$LEX14
cmp eax, ebx
jl If169End

push offset main$LEX19
CALL outputstr
If169End:

; String #54 :ivil@1
invoke GetHours, FindFactor$LEX1
push eax ;результат функции
pop mainnumber1

; String #55 :ivil@1
invoke GetMinutes, FindFactor$LEX1
push eax ;результат функции
pop mainnumber2

; String #56 :ivil@1
invoke GetDate, FindFactor$LEX1
push eax ;результат функции
pop mainnumber3

; String #57 :ivil@1
invoke GetMonth, FindFactor$LEX1
push eax ;результат функции
pop mainnumber4

push offset main$LEX24
CALL outputstr

push mainnumber1
CALL outputuint

push offset main$LEX25
CALL outputstr
```

```
push mainnumber2
CALL outputuint

push offset main$LEX26
CALL outputstr

push mainnumber3
CALL outputuint

push offset main$LEX27
CALL outputstr

push mainnumber4
CALL outputuint

mov eax, main$LEX28
     jmp endPoint
     div_by_0:
     push offset divideOnZeroExeption
CALL outputstr
endPoint:
     invoke         ExitProcess, eax
main ENDP
end main
```

Листинг 1 – Код исходной программы на языке Ассемблер