

МИНИСТЕРСТВО ОБРАЗОВАНИЯ И НАУКИ РОССИЙСКОЙ  
ФЕДЕРАЦИИ федеральное государственное бюджетное образовательное  
учреждение  
высшего образования  
«Тольяттинский государственный университет»

**Институт математики, физики и информационных технологий**

(наименование института полностью)

Кафедра «Прикладная математика и информатика»

(наименование кафедры полностью)

01.02.03 Прикладная математика и информатика

(код и наименование направления подготовки, специальности)

Компьютерные технологии и математическое моделирование

(направленность (профиль))

**КУРСОВАЯ РАБОТА**

по дисциплине (учебному курсу)

**Теоретические основы информатики**

(наименование дисциплины (учебного курса))

на тему «Разработка и реализация простейшего компилятора по заданному  
варианту исходных данных № 8»

Студент

Герасимов Н.Н.

(И.О. Фамилия)

(личная подпись)

Руководитель

Кузьмичев А.Б.

(И.О. Фамилия)

(личная подпись)

Оценка:

Дата:

«27» декабря 2018г

МИНИСТЕРСТВО ОБРАЗОВАНИЯ И НАУКИ РОССИЙСКОЙ  
ФЕДЕРАЦИИ

федеральное государственное бюджетное образовательное учреждение  
высшего образования  
«Тольяттинский государственный университет»

Институт математики, физики и информационных технологий  
(наименование института полностью)

Кафедра «Прикладная математика и информатика»  
(наименование кафедры полностью)

УТВЕРЖДАЮ

Зав. кафедрой

\_\_\_\_\_  
(подпись) (И.О. Фамилия)  
« \_\_\_\_ » \_\_\_\_\_ 20 \_\_\_\_ г.

**ЗАДАНИЕ**  
**на выполнение курсовой работы**

Студент 2 курса гр. ПМИп-1702а Герасимов Никита Николаевич

1. Тема: Разработка и реализация компилятора по заданному варианту исходных данных № 8
2. Срок сдачи студентом законченной курсовой работы: «27» декабря 2018г.
3. Исходные данные к курсовой работе: вариант № 8
4. Содержание курсовой работы (перечень подлежащих разработке вопросов, разделов)

Введение

1. Исходные данные для выполнения курсовой работы

2. Организация таблицы идентификаторов

3. Разработка лексического анализатора

4. Разработка синтаксического анализатора

5. Разработка генератора результирующего кода

Заключение

Список используемых источников

Приложение

5. Ориентировочный перечень графического и иллюстративного материала: пояснительная записка (25-35 стр.), презентация
6. Рекомендуемые учебно-методические материалы: Молчанов А. Ю. Системное программное обеспечение. Лабораторный практикум. – СПб.: Питер, 2005. – 284 с.: ил.
7. Дата выдачи задания «20» сентября 2018г.

Руководитель курсовой работы

\_\_\_\_\_  
(подпись)

Кузьмичев А.Б.

\_\_\_\_\_  
(И.О. Фамилия)

Задание принял к исполнению

\_\_\_\_\_  
(подпись)

Герасимов Н.Н.

\_\_\_\_\_  
(И.О. Фамилия)

## Календарный план

№	Этапы выполнения	% выполнения	Сроки выполнения по этапам	Отметка о выполнении этапа (дата. подпись)
1	Выбор способа описания таблицы идентификаторов и программная реализация выбранного способа	5%	21.09.18	
2	Построение лексического анализатора	25%	19.11.18	
3	Программная реализация лексического анализатора	5%	27.11.18	
4	Построение матрицы предшествования по заданной грамматики	5%	14.12.18	
5	Программная реализация синтаксического анализатора	30%	16.12.18	
6	Реализация генератора результатирующего кода	18%	20.12.18	
7	Отладка программы	2%	21.12.18	
8	Подготовка пояснительной записки	10%	22.12.18	

## Содержание

Введение.....	5
1. Исходные данные для выполнения курсовой работы .....	6
2. Организация таблицы идентификаторов .....	8
3. Разработка лексического анализатора .....	10
3.1 Описание лексического анализатора .....	10
3.2 Разработка лексического анализатора .....	13
4. Разработка синтаксического анализатора.....	17
4.1 Разработка матрицы предшествования.....	17
4.2 Разработка синтаксического анализатора .....	21
5. Разработка генератора результирующего кода.....	24
5.1 Описание генератора результирующего кода.....	24
5.2 Интегрирование разработанных модулей в компилятор.....	27
Заключение .....	30
Список используемых источников.....	32
Приложение А .....	33

## **Введение**

Целью курсовой работы является изучение составных частей, основных принципов построения и функционирования компиляторов, а также практическое освоение методов построения простейших компиляторов для заданной грамматики входного языка.

Конечным результатом курсовой работы является программная реализация заданного компилятора. Для реализации компилятора был выбран язык программирования C++.

Компилятор состоит из следующих частей:

- генератор таблицы идентификаторов;
- лексический анализатор;
- синтаксический анализатор;
- генератор результирующего кода.

Отсюда вытекают задачи курсовой работы:

- разработка генератора таблицы идентификаторов;
- разработка лексического анализатора;
- разработка синтаксического анализатора;
- разработка генератора результирующего кода.

## 1. Исходные данные для выполнения курсовой работы

Вариант 8: Входной язык содержит операторы цикла типа for (...; ...; ...) do, разделённые символом ; (точка с запятой). Операторы цикла содержат идентификаторы, знаки сравнения <, >, =, римские числа (M, D, C, L, X, V, I), знак присваивания (:=). [1, стр. 49]

Входной язык состоит из следующих лексем:

- ключевые слова: for, do;
- разделители: точка с запятой, открывающая и закрывающая круглые скобки;
- идентификаторы: произвольная последовательность букв латинского алфавита верхнего и нижнего регистров, арабских цифр от 0 до 9, знака подчёркивания, начинающаяся обязательно с буквы;
- римские числа: последовательность заглавных латинских букв I, V, X, L, C, D, M;
- знаки сравнения: <, >, =
- знак присваивания: :=

Римские числа составляются по следующим правилам:

- римские числа записываются при помощи повторения 7 символов: I, V, X, L, C, D, M;
- максимально возможное количество повторений одного символа – 3 раза;
- каждый символ имеет своё числовое значение: I = 1, V = 5, X = 10, L = 50, C = 100, D = 500, M = 1000;
- если большая цифра стоит перед меньшей, то они складываются (принцип сложения);
- если меньшая цифра стоит перед большей, то меньшая вычитается из большей (принцип вычитания);

- для обозначения чисел 4, 9, 40, 90, 400, 900 надо пользоваться принципом вычитания (5-1, 10-1, 50-10 и т. д.), а для всех остальных – принципом сложения (1+1+1, 5+1+1, 50+10+1+1 и т. д.);
- для правильной записи больших чисел римскими цифрами необходимо сначала записать число тысяч, затем сотен, затем десятков и, наконец, единиц.

Границами лексем служат пробел, знак табуляции, знак перевода строки и возврата каретки, разделители, идентификаторы, римские цифры, знаки сравнения и присваивания.

Комментарием в программе считается любая последовательность любых символов, заключённая между символами { и }.

Исходная грамматика в форме Бэкуса-Наура выглядит следующим образом:

$G(\{\text{for, do, a, :=, <, >, =, (, ), ;\}, \{S, F, T, E\}, \mathbf{P}, S)$  с правилами **P**:

$S \rightarrow F;$

$F \rightarrow \text{for } (T) \text{ do } F \mid \mathbf{a := a}$

$T \rightarrow F;E;F \mid ;E;F \mid F;E; \mid ;E;$

$E \rightarrow \mathbf{a < a} \mid \mathbf{a > a} \mid \mathbf{a = a}$

Символ S является начальным символом грамматики; S, F, T, E обозначают нетерминальные символы, а жирным шрифтом выделены терминальные символы.

В результате была описана исходная грамматика входного языка в форме Бэкуса-Наура. Входной язык содержит операторы цикла типа for (...; ...; ...) do, разделённые символом ; (точка с запятой). Операторы цикла содержат идентификаторы, знаки сравнения <, >, =, римские числа (M, D, C, L, X, V, I), знак присваивания (:=). Также описаны правила составления римских чисел и идентификаторов.

## 2. Организация таблицы идентификаторов

Перед выбором способа организации таблицы идентификаторов был проведён сравнительный анализ двух методов: хеш-адресация с рехешированием и упорядоченный список с применением сортировки Шелла.

Сравнительный анализ показал, что метод хеш-адресации эффективнее, потому что использует меньшее количество тактов процессора, а, следовательно, выполняется быстрее. На таблице 1 показана сравнительная характеристика двух методов.

Таблица 1 – Сравнительная характеристика двух методов

Количество идентификаторов	% заполнения таблицы	Количество тактов процессора		1-й метод / 2-й метод	2-й метод / 1-й метод
		Метод хеширования (1-й метод)	Сортировка Шелла (2-й метод)		
250	25	44106	42010	1,05	0,95
500	50	83635	90656	0,92	1,08
750	75	127369	145428	0,88	1,14
1000	100	197141	212839	0,93	1,08
Количество ячеек таблицы	1000				

На каждую операцию поиска элемента в таблице компилятор будет затрачивать время, а поскольку количество идентификаторов может быть велико, то очень важно организовать таблицу таким образом, чтобы поиск производился максимально быстро. Поэтому для организации таблицы идентификаторов был выбран способ с использованием хеш-адресации с рехешированием.

Данный способ основан на хеш-функции. Хеш-функция – это алгоритм, который на входе получает идентификатор, а затем при помощи математических операций преобразует его в целое число. Принцип действия данного метода заключается в следующем: хеш-функция вычисляет значение, это значение является номером ячейки в таблице, куда записывается идентификатор, если это место уже занято, то для рассмотрения берётся следующая ячейка. Также стоит отметить, что при одинаковых



идентификаторах значение хеш-функции будет одинаково, поэтому перед поиском свободной ячейки также проверяется на совпадение идентификатора с тем, что занесён в таблицу. Такое дополнение позволяет избежать дублирование идентификаторов.

Данный метод организации таблицы идентификаторов является наиболее эффективным, так как время проверки наличия такого же идентификатора зависит только от работы самой хеш-функции, это гораздо быстрее, чем многократное сравнение элементов таблицы.

В итоге был выбран способ организации таблицы идентификаторов на основе алгоритма хеш-адресации с рехешированием. Решение принято на основе сравнительного анализа данного метода построения таблицы с упорядоченным списком с применением сортировки Шелла. Результат показал, что метод рехеширования эффективнее, чем упорядоченный список с применением сортировки Шелла.

### 3. Разработка лексического анализатора

#### 3.1 Описание лексического анализатора

Процесс компиляции проходит в три этапа: лексический анализ -> синтаксический анализ -> генерация кода. Причём каждый этап зависит от предыдущего, то есть если в одном из них произошла ошибка, то прерывается весь этап компиляции.

Лексический анализ осуществляется при помощи лексического анализатора (далее ЛА). ЛА преобразует исходный текст программы в вектор структур, состоящих из двух элементов: `lexem` – строковая переменная, которая содержит лексему и `lexemType` – тип лексемы. Причём лексический анализатор игнорирует все символы, которые находятся между { и }.

Лексемы входного языка могут иметь следующие типы:

- Идентификаторы;
- Ключевые слова (`for`, `do`);
- Разделители (открывающая и закрывающая круглые скобки, точка с запятой)
- Знаки сравнения (больше, меньше, равно);
- Римские числа;
- Знак операции присваивания;

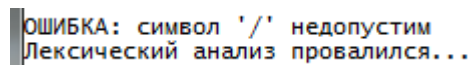
Лексический анализатор построен на основе конечного автомата (рисунки 2-4). Конечный автомат (далее КА) можно представить в виде ориентированного графа. Узлами графа являются состояния конечного автомата, а дуги – переход из одного состояния в другое. Подписи к дугам обозначают условие перехода, если при переходе требуется выполнить какое-либо действие, то это действие обозначается через | (вертикальную черту). На рисунках 2-4 используются следующие условные обозначения:

- S - начальное состояние

- F - функция для обработки данных в таблице лексем, которая может принимать в качестве аргумента a, v или d.
  - a - текущий символ
  - v - накопленная переменная
  - d - запоминает данные при переходе из одного состояния в другое и добавляет к имеющемуся новые данные
- Н - непечатаемые символы (перенос строки, возврат каретки, табуляция, пробел и т.д.)
- Л - буквы латинского алфавита
- Л(\*) - буквы латинского алфавита, кроме тех, что указаны в скобках
- Ц - арабские цифры (0-9)
- Р - символы римских цифр (I, V, X, L, C, D, M)
- Sk - пропуск текущего состояния и переход к тому состоянию, в котором встречается римская цифра

КА начинает алгоритм из состояния S, а завершает в состоянии F. Если в файле ещё остались символы, то он сразу переходит в начальное состояние S.

Если считался символ, который недопустим во входном языке, то в терминал выводится сообщение об ошибке и прерывается процесс компиляции (рисунок 1).



```
ОШИБКА: символ '/' недопустим
Лексический анализ провалился...
```

Рисунок 1 - Сообщение об ошибке на этапе лексического анализа

На рисунке 2 представлен граф КА для распознавания всех лексем, кроме ключевых слов и римских чисел. Ниже представлены графы КА для распознавания ключевых слов `for` и `do` (рисунок 3) и для распознавания римских чисел (рисунок 4).

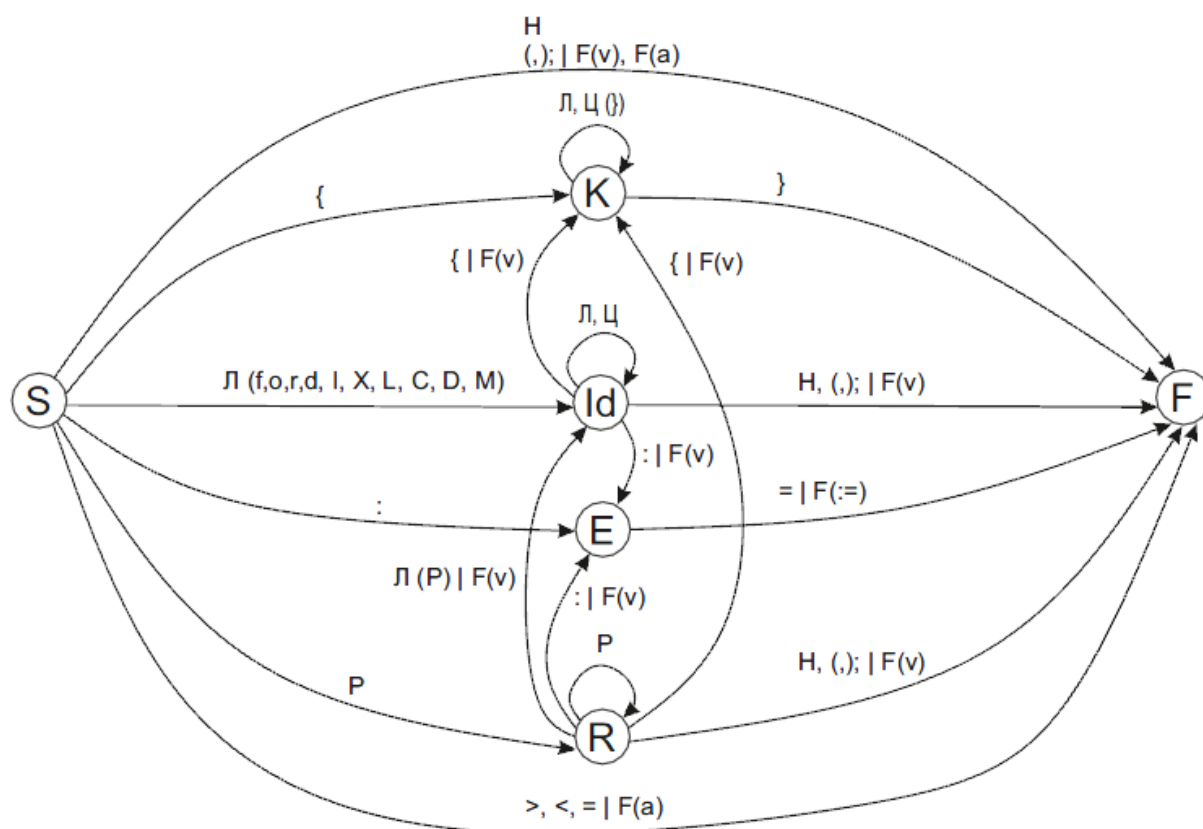


Рисунок 2 - Граф КА для распознавания лексем, исключая ключевые слова и римские числа

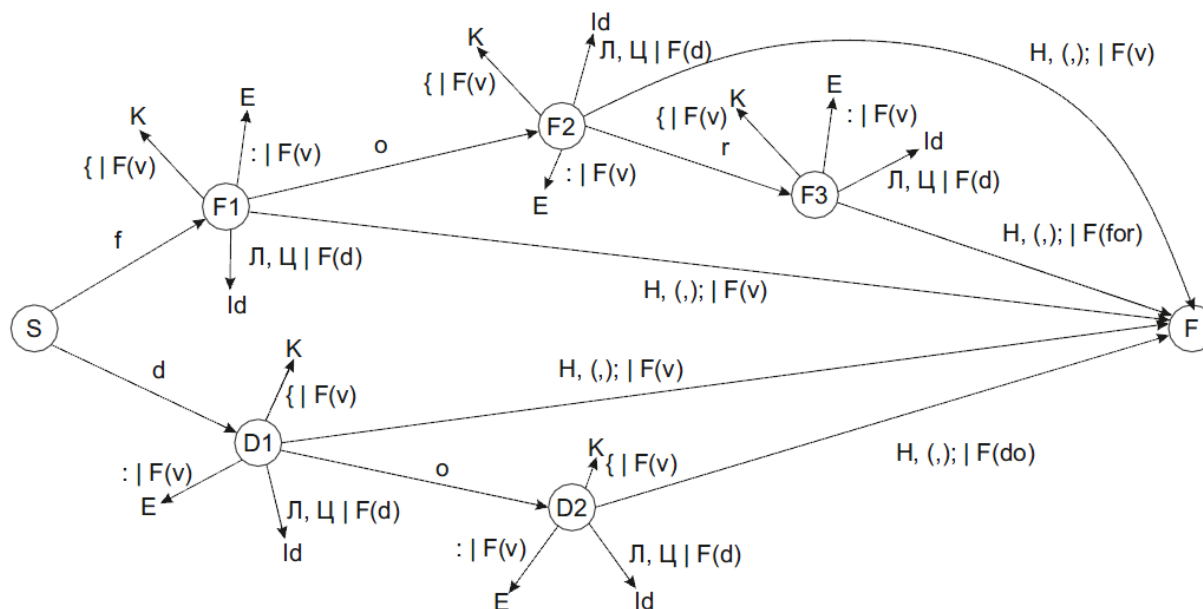


Рисунок 3 - Граф КА для распознавания ключевых слов for и do

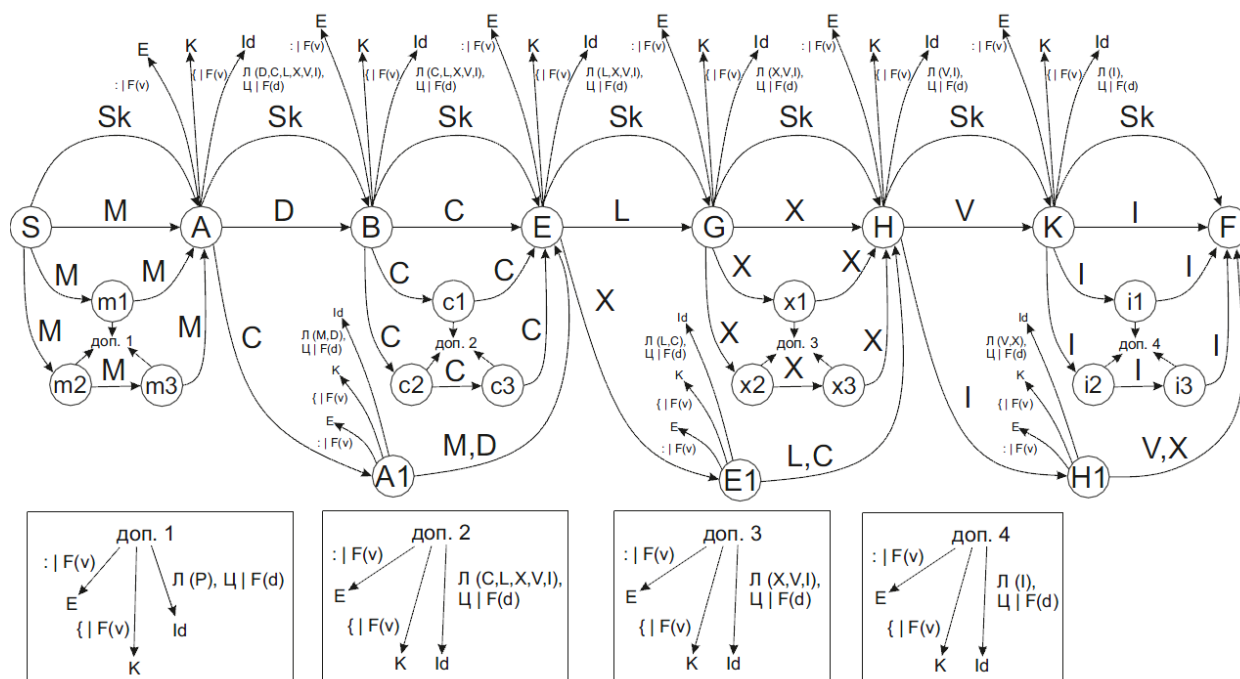


Рисунок 4 - Граф КА для распознавания римских чисел

В итоге, был описан конечный автомат лексического анализатора, он изображён на рисунка 2-4. Конечный автомат состоит из 29 состояний, включая промежуточные состояния ввода римских чисел, ключевых слов и других лексем.

### 3.2 Разработка лексического анализатора

Лексический анализатор начинает свой алгоритм с функции **lexems()**. Эта функция производит посимвольное считывание текста исходной программы из файла, который указал пользователь в качестве первого аргумента при запуске компилятора через терминал (по умолчанию считывание происходит из файла **main.txt**), до тех пор, пока не достигнет конца файла или пока не встретит ошибку. Принцип ее работы представлен в виде блок-схемы на рисунке 5.

После окончания проверки функция либо выведет сообщение об ошибке, либо вернет в главный файл полученные таблицы лексем и идентификаторов и перейдет к следующему этапу работы компилятора.

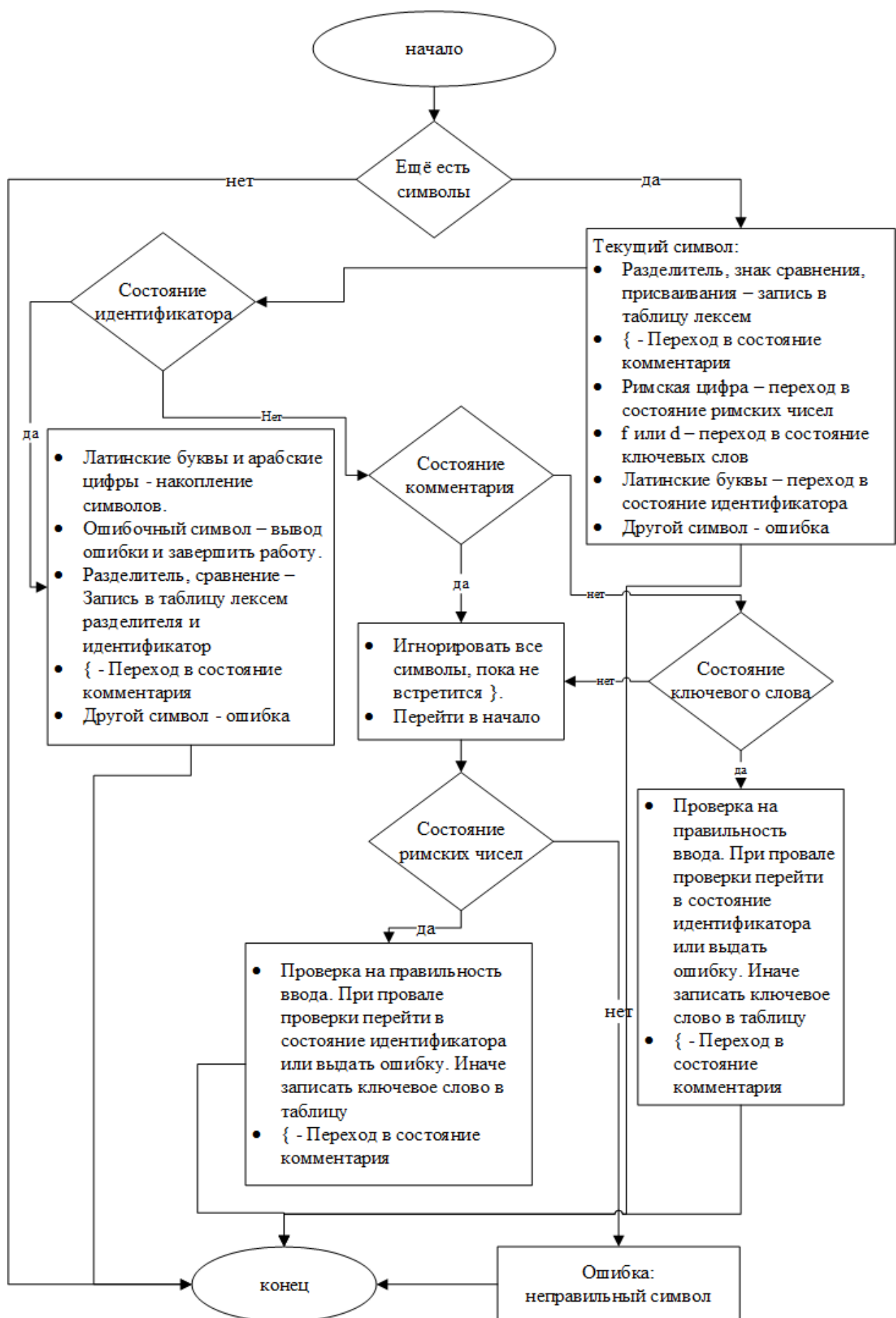


Рисунок 5 – Алгоритм выполнения лексического анализа

Если лексический анализ прошёл успешно, то будет сформирован файл, (по умолчанию название файла **lexem.txt**). Содержимое файла представлено в следующей форме: <ключ>/<лексема>. Каждый ключ обозначает соответствующий тип лексемы:

- s - разделитель
- c - знак сравнения
- i - идентификатор
- e - знак присваивания
- k - ключевое слово
- r - римское число

На рисунке 6 показано содержимое файла при успешном выполнении лексического анализа.

```
k/for
s/(
s/;
i/joge
c/<
r/xx
s/;
s/)
k/do
i/vox
e/:=
r/D
s/;
```

Рисунок 6 - Содержимое файла при успешном выполнении ЛА

Дальше выполняется подготовка к этапу синтаксического анализа: из файла, в который был выведен результат лексического анализа, происходит заполнение структуры <лексема>|<тип лексемы>, а также заполнение таблицы идентификаторов и формирование файла **tableID.txt**, в который выведена таблица идентификаторов (рисунок 7).

```
455. joge
195. vox
472. id1
137. ford
318. goks
237. mops
177. jjuf
32. gg
```

Рисунок 7 - Вывод таблицы идентификаторов в файл tableID.txt

В результате был описан алгоритм работы лексического анализатора. Если в ходе лексического анализа не было обнаружено ошибок, то формируется файл с разобранными лексемами. Результатом работы лексического анализатора является цепочка структур вида <лексема>|<тип лексемы> или сообщение об обнаруженной ошибке во входном файле.

Таким образом, был описан алгоритм работы лексического анализатора, который производит посимвольное считывание текста исходной программы из входного файла до тех пор, пока не достигнет конца файла или не встретит ошибку. Алгоритм посимвольного анализа представлен в виде блок-схемы на рисунке 5. Результатом работы лексического анализатора является цепочка структур вида <лексема>|<тип лексемы> или сообщение об обнаруженной ошибке во входном файле.



## 4. Разработка синтаксического анализатора

### 4.1 Разработка матрицы предшествования

Следующим этапом процесса компиляции является синтаксический анализ. Его основой является синтаксический анализатор, который имеет следующие задачи:

- Проверка правильности написания каждой синтаксической конструкции;
- Проверка принадлежности синтаксических конструкций в исходном тексте программы к входному языку;
- Представление синтаксических конструкций в виде, удобном для дальнейшей работы компилятора.

Для реализации синтаксического анализа использован алгоритм разбора «перенос - свертка». Для моделирования его работы необходима входная цепочка символов и стек символов, в котором автомат может обращаться не только к самому верхнему символу, но и к некоторой цепочке на вершине стека. Этот алгоритм для заданной КС-грамматики при наличии построенной матрицы предшествования описан в [2, стр. 74].

Прежде чем приступить к синтаксическому анализу, необходимо построить матрицу операторного предшествования. Исходная грамматика в форме Бэкуса-Наура выглядит следующим образом:

$G(\{\text{for, do, a, :=, <, >, =, (, ), ;\}, \{S, F, T, J\}, P, S)$  с правилами **P**:

$S \rightarrow F;$  - правило 1

$F \rightarrow \text{for } (T) \text{ do } F \mid a := a$  – правило 2, 3

$T \rightarrow F;E;F \mid ;E;F \mid F;E; \mid ;E;$  - правило 4, 5, 6, 7

$E \rightarrow a < a \mid a > a \mid a = a$  – правило 8, 9, 10

Эта грамматика является недостижимой, так как в правилах 3, 8, 9, 10 стоит три терминальных символа подряд. Поэтому надо её дополнить.

Конечная грамматика в форме Бэкуса-Наура выглядит следующим образом:

$G(\{\text{for, do, a, :=, <, >, =, (, ), ;\}, \{S, F, G, T, E, J\}, P, S)$  с правилами  $P$ :

$S \rightarrow F;$

$F \rightarrow \text{for } G \text{ do } F \mid \text{a} := J$

$G \rightarrow (T)$

$T \rightarrow F;E;F \mid ;E;F \mid F;E; \mid ;E;$

$E \rightarrow \text{a} < J \mid \text{a} > J \mid \text{a} = J$

$J \rightarrow \text{a}$

Теперь необходимо построить множество крайних правых и крайних левых символов. На первом шаге берутся все крайние левые и крайние правые символы из правил конечной грамматики  $G$ . Полученное множество представлено в таблице 2.

Таблица 2 - Множество крайних правых и левых символов. Шаг 1

Символ $U$	$L(U)$	$R(U)$
$S$	$F$	$;$
$F$	$\text{for, a}$	$F, J$
$G$	$($	$)$
$T$	$F, ;$	$F, ;$
$E$	$\text{a}$	$J$
$J$	$\text{a}$	$\text{a}$

Эта таблица не является конечной, так как множества  $L(U)$  для символов  $S, T$  и множества  $R(U)$  для символа  $F, T, E$  содержат другие терминальные символы, поэтому они должны быть дополнены. Результат представлен в таблице 3.

Таблица 3 – Множество крайних правых и левых символов. Шаг 2

Символ $U$	$L(U)$	$R(U)$
$S$	$F, \text{for, a}$	$;$
$F$	$\text{for, a}$	$F, J, \text{a}$
$G$	$($	$)$
$T$	$F, ;, \text{for, a}$	$F, ;, J, \text{a}$
$E$	$\text{a}$	$J, \text{a}$
$J$	$\text{a}$	$\text{a}$

Теперь необходимо построить множество крайних правых и крайних левых терминальных символов. На первом шаге возьмём все крайние левые и крайние правые терминальные символы из конечной грамматики  $G$ . Полученное множество представлено в таблице 4.

Таблица 4 – Множество крайних правых и левых терминальных символов.

Шаг 1

Символ $U$	$L(U)$	$R(U)$
S	;	;
F	for, a	do, :=
G	(	)
T	;	;
E	a	<, >, =
J	a	a

Эта таблица также не является конечной. Теперь необходимо дополнить множество на основании ранее построенного множества крайних левых и крайних правых символов в таблице 3. Результат представлен в таблице 5.

Таблица 5 – Множество крайних правых и левых терминальных символов.

Шаг 2

Символ $U$	$L(U)$	$R(U)$
S	for, a, ;	;
F	for, a	a, do, :=;
G	(	)
T	for, a, ;	a, do, :=,;
E	a	<, >, =

Для заполнения матрицы операторного предшествования используются множества крайних левых и крайних правых терминальных символов и правила конечной грамматики  $G$ . Полученная матрица операторного предшествования представлена в таблице 6.

Таблица 6 – Матрица операторного предшествования

Символы	for	do	a	:=	<	>	=	(	)	;	$\perp_k$
for		=·						<·			
do	<·		<·						·>	·>	
a				=·	=·	=·	=·		·>	·>	
:=			<·						·>	·>	
<			<·							·>	
>			<·							·>	
=			<·							·>	
(	<·		<·						=·	<·	
)		·>									
;	<·		<·						·>	=·	·>
$\perp_n$	<·		<·							<·	

Отношения предшествования обозначаются знаками «=·», «<·» и «·>». Отношение предшествования зависит от того, в каком порядке стоят символы. Если между терминальными символами нет отношения, то ячейка матрицы остаётся пустой.

На основе конечной грамматики G была построена остовная грамматика:

$G'(\{\text{for, do, a, :=, <, >, =, (, ), ;\}, \{E\}, P', S)$  с правилами P':

$E \rightarrow E;$  - правило 1

$E \rightarrow \text{for } E \text{ do } E \mid a := E$  - правило 2, 3

$E \rightarrow (E)$  – правило 4

$E \rightarrow E;E;E \mid ;E;E \mid E;E; \mid ;E;$  - правило 5, 6, 7, 8

$E \rightarrow a<E \mid a>E \mid a=E$  – правило 9, 10, 11

$E \rightarrow a$  – правило 12

Таким образом была описана разработка матрицы операторного предшествования для реализации синтаксического анализа. Данная матрица представлена в таблице 6. Для разработки матрицы операторного предшествования была создана таблица множества крайних правых и крайних левых терминальных символов. Матрица предшествования позволяет показать всевозможные отношения терминальных символов в программе входного языка.

## 4.2 Разработка синтаксического анализатора

Синтаксический анализатор реализован в виде автомата с магазинной памятью (далее МП-автомат). Перед началом разбора формируется входная цепочка из структур лексем, стек и список правил, затем в конец входной цепочки помещается конечный символ, а в стек помещается начальный символ. Далее выполняется цикл, алгоритм которого изображён на рисунке 8.

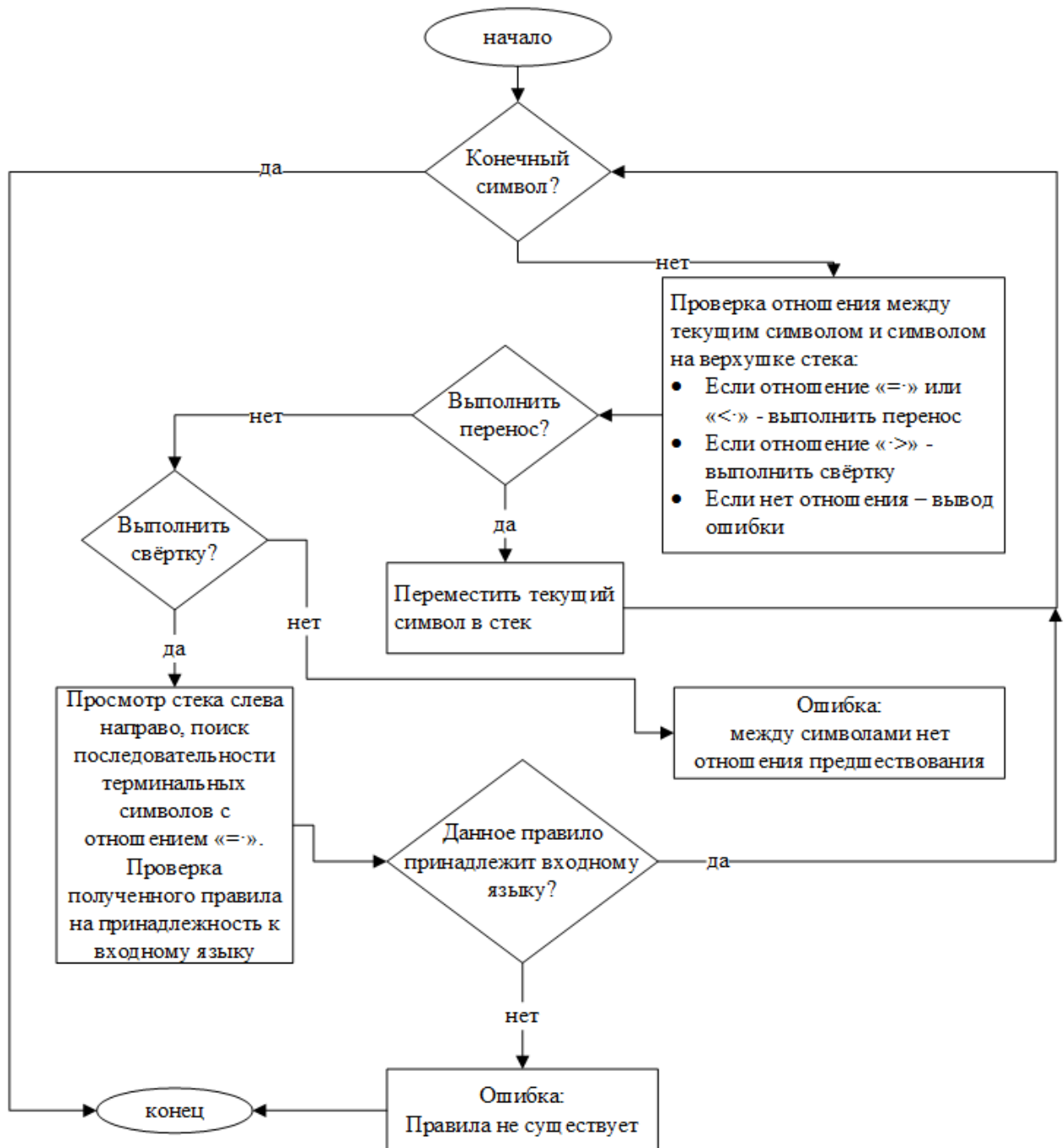


Рисунок 8 – Алгоритм синтаксического анализа

В процессе синтаксического анализа формируется файл (по умолчанию название файла **syntax.txt**). В этом файле записывается состояние автомата на каждом шаге в следующем виде {a|b|y} (<действие>), где

- a - непрочитанная часть входной цепочки
- b - содержимое стека МП-автомата
- y - последовательность применённых правил
- действие – перенос или свёртка

На рисунке 9 показано содержимое данного файла.

```
<for < joge := I ; joge < XX ; joge := vox > do vox := D ; K ! H ! > <перенос>
<< joge := I ; joge < XX ; joge := vox > do vox := D ; K ! H for ! > <перенос>
<joge := I ; joge < XX ; joge := vox > do vox := D ; K ! H for < ! > <перенос>
<:= I ; joge < XX ; joge := vox > do vox := D ; K ! H for < joge ! > <перенос>
<I ; joge < XX ; joge := vox > do vox := D ; K ! H for < joge := ! > <перенос>
<; joge < XX ; joge := vox > do vox := D ; K ! H for < joge := I ! > <свертка>
<; joge < XX ; joge := vox > do vox := D ; K ! H for < joge := E ! 11 > <свертка>
<; joge < XX ; joge := vox > do vox := D ; K ! H for < E ! 11 2 > <перенос>
<joge < XX ; joge := vox > do vox := D ; K ! H for < E ; ! 11 2 > <перенос>
<< XX ; joge := vox > do vox := D ; K ! H for < E ; joge ! 11 2 > <перенос>
<XX ; joge := vox > do vox := D ; K ! H for < E ; joge < ! 11 2 > <перенос>
<; joge := vox > do vox := D ; K ! H for < E ; joge < XX ! 11 2 > <свертка>
<; joge := vox > do vox := D ; K ! H for < E ; joge < E ! 11 2 11 > <свертка>
<; joge := vox > do vox := D ; K ! H for < E ; E ! 11 2 11 8 > <перенос>
<joge := vox > do vox := D ; K ! H for < E ; E ; ! 11 2 11 8 > <перенос>
<:= vox > do vox := D ; K ! H for < E ; E ; joge ! 11 2 11 8 > <перенос>
<vox > do vox := D ; K ! H for < E ; E ; joge := ! 11 2 11 8 > <перенос>
<> do vox := D ; K ! H for < E ; E ; joge := vox ! 11 2 11 8 > <свертка>
<> do vox := D ; K ! H for < E ; E ; joge := E ! 11 2 11 8 11 > <свертка>
<> do vox := D ; K ! H for < E ; E ; E ! 11 2 11 8 11 2 > <свертка>
<> do vox := D ; K ! H for < E ! 11 2 11 8 11 2 4 > <перенос>
<do vox := D ; K ! H for < E > ! 11 2 11 8 11 2 4 > <свертка>
<do vox := D ; K ! H for E ! 11 2 11 8 11 2 4 3 > <перенос>
<vox := D ; K ! H for E do ! 11 2 11 8 11 2 4 3 > <перенос>
<:= D ; K ! H for E do vox ! 11 2 11 8 11 2 4 3 > <перенос>
<D ; K ! H for E do vox := ! 11 2 11 8 11 2 4 3 > <перенос>
<; K ! H for E do vox := D ! 11 2 11 8 11 2 4 3 > <свертка>
<; K ! H for E do vox := E ! 11 2 11 8 11 2 4 3 11 > <свертка>
<; K ! H for E do E ! 11 2 11 8 11 2 4 3 11 2 > <свертка>
<; K ! H E ! 11 2 11 8 11 2 4 3 11 2 1 > <перенос>
<K ! H E ; ! 11 2 11 8 11 2 4 3 11 2 1 > <свертка>
<K ! H E ! 11 2 11 8 11 2 4 3 11 2 1 0 > <конец>
```

Рисунок 9 – Содержимое файла syntax.txt

Также предусмотрены случаи, если в исходной программе встречается последовательность терминальных символов, которые не связаны отношением предшествования (рисунок 10), или если неправильно введена синтаксическая конструкция (рисунок 11).

ОШИБКА: между ; и ) нет отношения предшествования.  
Синтаксический анализ провалился...

Рисунок 10 – Сообщение об ошибке

ОШИБКА: такого правила не существует: E ; E ; E ; E ;

Рисунок 11 – Сообщение об ошибке

В результате была описана программная реализация синтаксического анализатора. Синтаксический анализатор на входе получает цепочку из лексем, осуществляет проверку на правильность синтаксических конструкций, затем преобразует их в упорядоченный список правил.

Таким образом, в разделе 4 был описан процесс разработки матрицы операторного предшествования и программная реализация синтаксического анализатора. Синтаксический анализатор получает на входе цепочку из лексем, осуществляет проверку на правильность синтаксических конструкций, используя матрицу операторного предшествования, затем преобразует их в упорядоченный список правил. Синтаксический анализ выполняется по алгоритму «перенос-свёртка». Данный алгоритм представлен в виде блок-схемы на рисунке 8.

## 5. Разработка генератора результирующего кода

### 5.1 Описание генератора результирующего кода

Если синтаксический анализ прошёл успешно, то программа переходит к последнему этапу компиляции – генерации кода. В качестве выходного языка был выбран Ассемблер. На текущем этапе компиляции в программе сформирована цепочка структур вида <номер правила>|<лексема>. На рисунке 12 показано содержание данной цепочки при разборе конструкции `for (joge := I; joge < XX; joge := vox) do vox:=D;`

```
Правило № 1 Лексема: E;  
Правило № 2 Лексема: forEdoE  
Правило № 3 Лексема: vox:=500  
Правило № 12 Лексема: 500  
Правило № 4 Лексема: (E)  
Правило № 5 Лексема: E;E;E  
Правило № 3 Лексема: joge:=vox  
Правило № 12 Лексема: vox  
Правило № 9 Лексема: joge<20  
Правило № 12 Лексема: 20  
Правило № 3 Лексема: joge:=1  
Правило № 12 Лексема: 1
```

Рисунок 12 – Список правил при разборе текста исходной программы

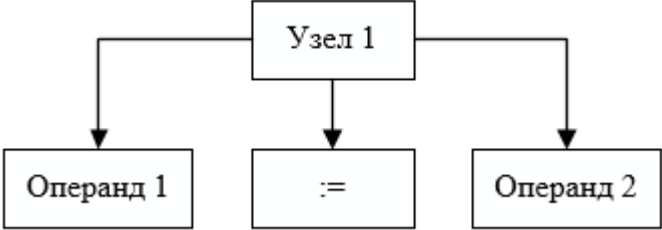

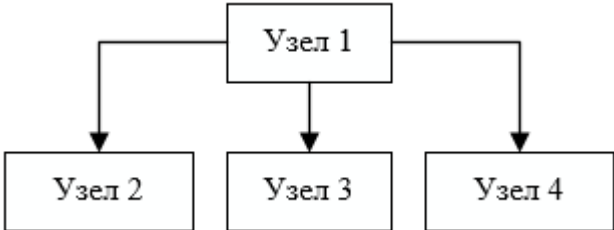
Также на этом этапе происходит конвертация римских чисел в арабские числа. Это сделано для удобства перевода кода в Ассемблер.

Исходный язык достаточно примитивный, в нём возможно всего два варианта программ: одиночное действие (например, `gg:=CCLX;`) или цикл с единственным действием (например, `for(joge:=I;joge<XX;joge:=vox)do vox:=D;`). При вводе других конструкций будет либо выведена ошибка на этапе синтаксического анализа, либо выведено сообщение о нарушении логики программы.

Затем производится обход цепочки структур вида <номер правила>|<лексема> и параллельная запись в выходной файл программного кода на языке Ассемблер согласно таблице 7.

Таблица 7 – Разбор узлов дерева



Вид узла дерева	Результирующий код	Примечание
 <pre> graph TD     Узел1[Узел 1] --&gt; Операнд1[Операнд 1]     Узел1 --&gt; Assign[:=]     Узел1 --&gt; Операнд2[Операнд 2] </pre>	<pre> mov  eax, DWORD PTR [rbp-4] mov  DWORD PTR [rbp-8], eax </pre>	<p>mov – операция присваивания</p> <p>DWORD PTR [rbp-&lt;число&gt;] – указатель на область памяти хранения переменной</p>
 <pre> graph TD     Узел1[Узел 1] --&gt; Операнд1[Операнд 1]     Узел1 --&gt; Compare[сравнение]     Узел1 --&gt; Операнд2[Операнд 2] </pre>	<pre> cmp  DWORD PTR [rbp-4], DWORD PTR [rbp-8] ja   .exit: </pre>	<p>cmp – операция сравнения</p> <p>DWORD PTR [rbp-&lt;число&gt;] – указатель на область памяти хранения переменной</p> <p>сравнение – знак сравнения, на этом месте может стоять &lt;, &gt; или =</p> <p>В зависимости от знака сравнения ставится указатель перехода в случае невыполнения условия:  ja – если стоит &lt;  jl – если стоит &gt;  jne – если стоит =</p>
 <pre> graph TD     Узел1[Узел 1] --&gt; Узел2[Узел 2]     Узел1 --&gt; Узел3[Узел 3]     Узел1 --&gt; Узел4[Узел 4] </pre>	<pre> Code(Узел 2) .for: Code(Узел 3) Code(Узел 4) jmp  .for: </pre>	<p>Узел 2 – присвоение начального значения переменной</p> <p>Code(Узел 2) – код, порождаемый процедурой для</p>

Продолжение таблицы 7

Вид узла дерева	Результирующий код	Примечание
		<p>нижележащего узла</p> <p>Узел 3 – условие выхода из цикла Code(Узел 3) – код, порождаемый процедурой для нижележащего узла</p> <p>Узел 4 – операции, выполняющиеся в теле цикла Code(Узел 4) – код, порождаемый процедурой для нижележащего узла</p>

На рисунке 13 показан пример дерево разбора для входной программы *for (joge:=I; joge<XX; joge:=vox)do vox:=D;*

А на рисунке 14 представлено содержимое выходного файла.

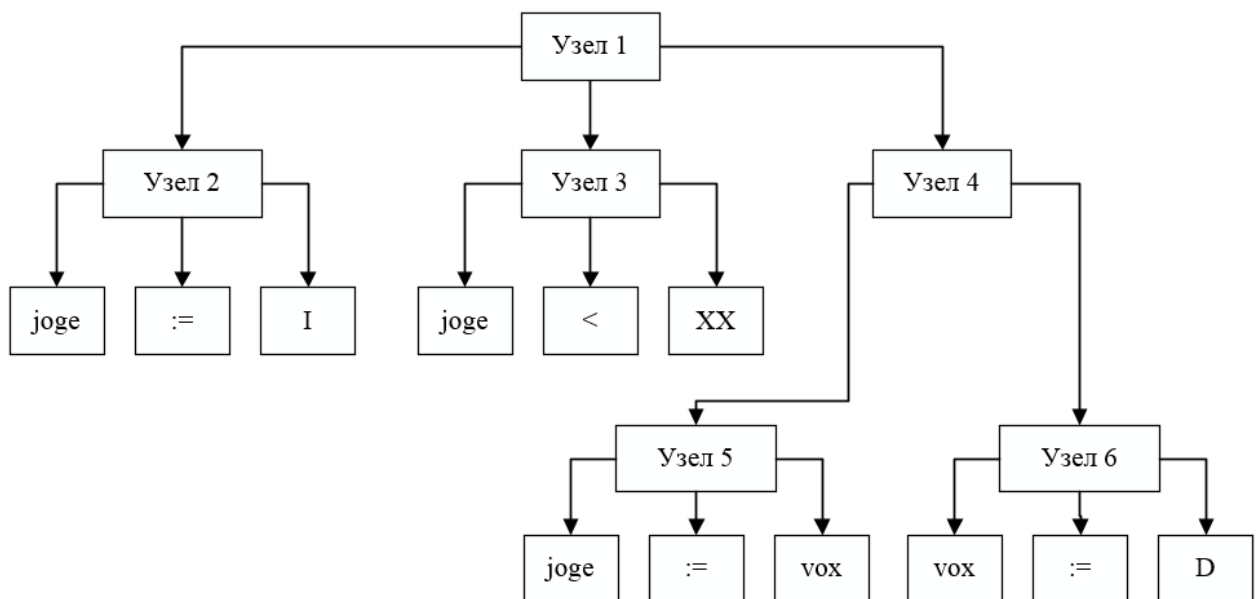


Рисунок 13 – Дерево разбора входной программы

```

.main:
push    rbp
mov     rbp, rsp
mov     eax, DWORD PTR [rbp-4]
mov     1, eax

.for:
cmp     DWORD PTR [rbp-4], 20
ja      .exit:
mov     eax, DWORD PTR [rbp-4]
mov     DWORD PTR [rbp-8], eax
mov     eax, DWORD PTR [rbp-8]
mov     500, eax
jmp     .for:

.exit:
mov     eax, 0
pop     rbp
ret

```

Рисунок 14 – Содержимое выходного файла

Таким образом, в процессе последнего этапа компиляции – генерации кода – компилятор конвертирует римские числа в арабские, выполняет дерево разбора и генерирует код исходной программы на языке Ассемблер.

## 5.2 Интегрирование разработанных модулей в компилятор

Компилятор состоит из главного файла программы **main.cpp**, к которому подключены три модуля. Каждый модуль описан в отдельном файле и имеет соответствующее название:

- лексический анализатор – **lexem.hpp**
- синтаксический анализатор – **synt.hpp**
- генератор результирующего кода – **generation.hpp**

Схематичное изображение структуры программы представлено на рисунке 15. Если очередной этап компиляции прошёл успешно, то об этом выводится сообщение в терминал. Если в каком-либо из этапов допускается ошибка, то прекращается работа всей программы.

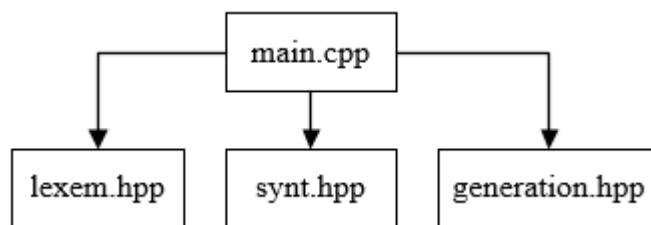


Рисунок 15 – Структура компилятора

В процессе компиляции формируются промежуточные файлы (рисунок 16). В этих файлах содержатся данные, которые были сформированы в ходе компиляции на определённом этапе.

По умолчанию файлы имеют такие названия:

- main.txt – файл исходной программы
- lexem.txt – файл со списком лексем и их типами
- syntax.txt – файл синтаксического разбора
- assembler.txt – выходной файл программы
- tableID.txt – файл со списком идентификаторов

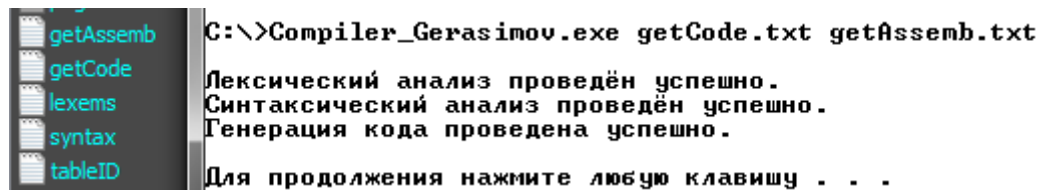
assembler	txt	262
lexems	txt	115
main	txt	42
syntax	txt	2 313
tableID	txt	21

Рисунок 16 – Список промежуточных файлов

Пользователь может управлять названиями файлов через терминал. Предусмотрено два способа:

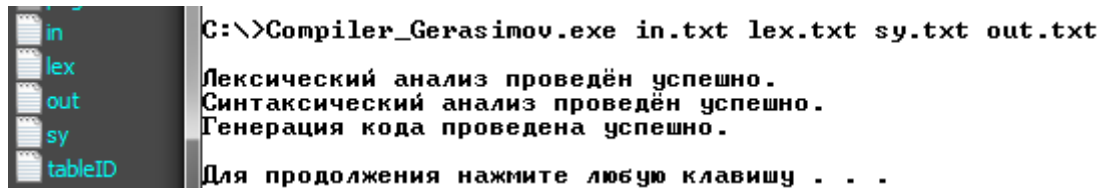
1. Вводится два параметра: первый параметр отвечает за название входного файла, второй параметр отвечает за название выходного файла (рисунок 17).
2. Вводится четыре параметра: первый – название входного файла, второй – название файла со списком лексем, третий – название файла синтаксического разбора, четвёртый – выходной файл (рисунок 18).

Если будет введено иное количество параметров, то будут использованы параметры по умолчанию.



```
C:\>Compiler_Gerasimov.exe getCode.txt getAssemb.txt
Лексический анализ проведён успешно.
Синтаксический анализ проведён успешно.
Генерация кода проведена успешно.
Для продолжения нажмите любую клавишу . . .
```

Рисунок 17 – Результат программы с двумя параметрами



```
C:\>Compiler_Gerasimov.exe in.txt lex.txt sy.txt out.txt
Лексический анализ проведён успешно.
Синтаксический анализ проведён успешно.
Генерация кода проведена успешно.
Для продолжения нажмите любую клавишу . . .
```

Рисунок 18 – Результат программы с четырьмя параметрами

В результате описан процесс интеграции разработанных модулей в компилятор. Компилятор состоит из главного файла, к которому подключены три модуля. Каждый модуль описан в отдельном файле. Компилятор запускается терминалом с двумя или четырьмя входными параметрами. Если будет введено иное количество параметров или параметры не будут введены, то компилятор будет использовать параметры по умолчанию.

Таким образом, в разделе 5 описан процесс генерации результирующего кода при помощи перевода текста исходной программы на язык Ассемблер.

## **Заключение**

Целью курсовой работы являлось получение теоретических знаний и овладение навыками разработки простейших компиляторов.

В разделе 1 описана исходная грамматика входного языка в форме Бэкуса-Наура, список возможных лексем, правила составления римских чисел и идентификаторов.

В разделе 2 был описан способ организации таблицы идентификаторов на основе алгоритма хеш-адресации с рехешированием. Решение принято на основе сравнительного анализа данного метода построения таблицы с упорядоченным списком с применением сортировки Шелла. Результат показал, что метод рехеширования эффективнее, чем упорядоченный список с применением сортировки Шелла.

В подразделе 3.1 описан созданный конечный автомат лексического анализатора, конечный автомат изображён на рисунках 3-5. Полученный автомат состоит из 29 состояний, включая промежуточные состояния ввода римских чисел, ключевых слов и других лексем.

В подразделе 3.2 описан алгоритм работы лексического анализатора. Если в ходе лексического анализа не было обнаружено ошибок, то формируется файл с разобранными лексемами. Результатом работы лексического анализатора является цепочка структур вида <лексема>|<тип лексемы> или сообщение об обнаруженной ошибке во входном файле.

В разделе 3 описан алгоритм работы лексического анализатора, который производит посимвольное считывание текста исходной программы из входного файла до тех пор, пока не достигнет конца файла или не встретит ошибку. Алгоритм посимвольного анализа представлен в виде блок-схемы на рисунке 5.

В подразделе 4.1 описана разработка матрицы операторного предшествования для реализации синтаксического анализа. Данная матрица представлена в таблице 6. Для разработки матрицы операторного

предшествования была создана таблица множества крайних правых и крайних левых терминальных символов. Матрица предшествования позволяет показать всевозможные отношения терминальных символов в программе входного языка.

В подразделе 4.2 описана программная реализация синтаксического анализатора, который из входной цепочки лексем составляет синтаксические конструкции, проверяет их на принадлежность ко входному языку и составляет упорядоченный список применённых правил.

В разделе 4 описан процесс разработки матрицы операторного предшествования и программная реализация синтаксического анализатора. Синтаксический анализ выполняется по алгоритму «перенос-свёртка». Данный алгоритм представлен в виде блок-схемы на рисунке 8

В подразделе 5.1 описан процесс перевода текста исходной программы на язык Ассемблер. Компилятор конвертирует римские числа в арабские выполняет дерево разбора и генерирует код исходной программы на языке Ассемблер.

В подразделе 5.2 описан процесс интеграции разработанных модулей в компилятор. Компилятор состоит из главного файла, к которому подключены три модуля. Каждый модуль описан в отдельном файле. Также описаны возможности управления пользователем процессом компиляции при запуске компилятора через терминал.

В разделе 5 описан процесс генерации результирующего кода при помощи перевода текста исходной программы на язык Ассемблер.

Результатом выполнения курсовой работы является программа, выполняющая проверку исходного текста программы по заданной грамматике и осуществляющая перевод программы на язык Ассемблера.

С исходным текстом программы можно ознакомиться в Приложении А.

## **Список используемых источников**

1. Молчанов А. Ю. Системное программное обеспечение: Учебник для вузов. 3-е изд. – СПб.: Питер, 2010. – 400 с.: ил.
2. Молчанов А. Ю. Системное программное обеспечение. Лабораторный практикум. – СПб.: Питер, 2005. – 284 с.: ил.
3. Compiler Explorer is an interactive online compiler which shows the assembly output of compiled C++, Rust, Go (and many more) code. [Электронный ресурс]: Режим доступа: <https://godbolt.org/>, 2018.
4. Asmworld. Программирование на ассемблере для начинающих и не только. [Электронный ресурс]: Режим доступа: <http://asmworld.ru/uchebnyj-kurs/016-uslovnye-i-bezuslovnye-perexody/>, 2010.



## Приложение А

Приложение А включает в себя диск, на котором находятся две папки. В папке **Проект CodeBlocks** содержится проект для CodeBlocks::IDE. В папке **Исходные файлы** содержатся файлы самого компилятора (main.cpp, lexem.hpp, synt.hpp, generation.hpp), скомпилированная программа под операционную систему Windows в формате .exe (Compiler\_Gerasimov.exe) и текст входной программы, дерево разбора которой было изображено на рисунке 13 (main.txt).