МИНИСТЕРСТВО ОБРАЗОВАНИЯ И НАУКИ РОССИЙСКОЙ ФЕДЕРАЦИИ
ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ БЮДЖЕТНОЕ ОБРАЗОВАТЕЛЬНОЕ УЧРЕЖДЕНИЕ ВЫСШЕГО ОБРАЗОВАНИЯ
«НИЖЕГОРОДСКИЙ ГОСУДАРСТВЕННЫЙ ТЕХНИЧЕСКИЙ УНИВЕРСИТЕТ ИМ. Р.Е. АЛЕКСЕЕВА»

Институт радиоэлектроники и информационных технологий

Кафедра «Прикладная математика»

Дисциплина: «Теория компиляции»

Курсовая работа на тему:

Проектирование и реализация систем лексического и синтаксического анализа программного кода

Выполнил:

Студент гр. 22-ПМ-1 _____ Зырянов Е.А. _

(группа) *(подпись)* *(Ф.И.О.)*

Проверил:

ассистент каф. ПМ _____ Кокоулина М.В.

(подпись) *(Ф.И.О.)*

Защищено с оценкой: _____

Дата защиты:

Содержание

Введение

Несмотря на более чем полувековую историю вычислительной техники, формально годом рождения теории компиляторов можно считать 1957, когда появился первый компилятор языка Фортран, созданный Бэкусом и дающий достаточно эффективный объектный код. До этого времени создание компиляторов было весьма «творческим» процессом. Лишь появление теории формальных языков и строгих математических моделей позволило перейти от «творчества» к «науке». Именно благодаря этому, стало возможным появление сотен новых языков программирования.

Несмотря на то, что к настоящему времени разработаны тысячи различных языков и их компиляторов, процесс создания новых приложений в этой области не прекращается. Это связно как с развитием технологии производства вычислительных систем, так и с необходимостью решения все более сложных прикладных задач.[1] Такая разработка может быть обусловлена различными причинами, в частности, функциональными ограничениями, отсутствием локализации, низкой эффективностью существующих компиляторов. Поэтому основы теории языков и формальных грамматик, а также практические методы разработки компиляторов лежат в фундаменте инженерного образования по информатике и вычислительной технике.

Целью курсовой работы является закрепление теоретических знаний в области теории формальных языков, грамматик, автоматов и методов трансляции, формирование практических умений и навыков разработки собственного компилятора модельного языка программирования, закрепление практических навыков самостоятельного решения инженерных задач.

Глава 1. Постановка задачи

Разработать компилятор модельного языка, выполнив следующие действия.

1) В соответствии с номером варианта составить формальное описание модельного языка программирования с помощью:

а) РБНФ;

б) диаграмм Вирта;

в) формальных грамматик.

2) Написать пять содержательных примеров программ, раскрывающих особенности конструкций учебного языка программирования, отразив в этих примерах все его функциональные возможности.

3) Составить таблицы лексем и диаграмму состояний с действиями для распознавания и формирования лексем языка.

4) По диаграмме с действиями написать функцию сканирования текста входной программы на модельном языке.

5) Разработать программное средство, реализующее лексический анализ текста программы на входном языке.

6) Реализовать синтаксический анализатор текста программы на модельном языке методом рекурсивного спуска.

7) Построить цепочку вывода и дерево разбора простейшей программы на модельном языке из начального символа грамматики.

Глава 2. Представление языка

Вариант модельного языка – 121132

В соответствии с вариантом был получен модельный язык, конструкции которого описаны ниже в расширенной форме Бекуса-Наура.

Существуют три основных метода описания синтаксиса языков программирования: формальные грамматики, формы Бэкуса-Наура и диаграммы Вирта[2].

Формы Бэкуса-Наура (БНФ)

Метаязык, предложенный Бэкусом и Науром, использует следующие обозначения: - символ «::=» отделяет левую часть правила от правой (читается: «определяется как»);

нетерминалы обозначаются произвольной символьной строкой, заключенной в угловые скобки «<» и «>»; - терминалы - это символы, используемые в описываемом языке;

правило может определять порождение нескольких альтернативных цепочек, отделяемых друг от друга символом вертикальной черты «|» (читается: «или»).

Расширенные формы Бэкуса-Наура (РБНФ)

Для повышения удобства и компактности описаний, в РБНФ вводятся следующие дополнительные конструкции (метасимволы):

квадратные скобки «[» и «]» означают, что заключенная в них синтаксическая конструкция может отсутствовать;

фигурные скобки «{» и «}» означают повторение заключенной в них синтаксической конструкции ноль или более раз;

сочетание фигурных скобок и косой черты «{/» и «/}» используется для обозначения повторения один и более раз;

круглые скобки «(» и «)» используются для ограничения альтернативных конструкций.

Синтаксис группы операций

<операции_группы_отношения>:: = < > | = | < | <= | > | >=

Операции группы «сложение»

<операции_группы_сложения>:: = + | - | or

- Операции группы «умножение»

<операции_группы_умножения>::= * | / | and

Унарная операция

<унарная_операция>::= not

Структура программы

<программа>::= «{» {/ (<описание> | <оператор>) ; /} «}»

Синтаксис команд описания данных

<описание>::= {<идентификатор> {, <идентификатор> } : <тип> ;}

Описание типов

(в порядке следования: целый, действительный, логический)

<тип>::= % | ! | $

Синтаксис составного оператора

<составной>::= «{» <оператор> { ; <оператор> } «}»

Синтаксис оператора присваивания

<присваивания> ::= <идентификатор> = <выражение>

Синтаксис оператора условного перехода

<условный>::= if <выражение> then <оператор> [else <оператор>]

end_else

Синтаксис оператора цикла с фиксированным числом повторений

<фиксированного_цикла>::= for ([<выражение>] ; [<выражение>] ; [<выражение>]) <оператор>

Синтаксис условного оператора цикла

<условного_цикла>::= do while <выражение> <оператор> loop

Синтаксис оператора ввода

<ввода>::= input (<идентификатор> {пробел <идентификатор>})

Синтаксис оператора вывода

<вывода>::= output (<выражение> { пробел <выражение> })

Синтаксис многострочных комментариев

Признак начала комментария /*

Признак конца комментария */

Примеры программ:

1. Сложение и вычитание 2х чисел.

```
{
 /* Описание переменных */
 num1, num2, sum, difference: %;

 /* Ввод значений */
 input(num1 num2);

 /* Вычисление суммы и разности */
 sum = num1 + num2;
 difference = num1 - num2;

 /* Вывод результатов */
 output(sum);
 output(difference);
}
```

2. Поиск максимального числа.

```
{
 /* Описание переменных */
 num1, num2, num3, max: %;

 /* Ввод значений */
 input(num1 num2 num3);

 /* Поиск максимального числа */
 max = num1;
 if num2 > max then
   max = num2;
 end_else
 if num3 > max then
   max = num3;
 end_else

 /* Вывод результата */
 output(max);
}
```

3. Проверка на четность.

```
{
 /* Описание переменной */
 num: %;
 is_even: $;

 /* Ввод значения */
 input(num);

 /* Проверка на четность */
 is_even = false;
 if num % 2 = 0 then
   is_even = true;
 end_else

 /* Вывод результата */
 output(is_even);
}
```

4. Вычисление факториала.

```
{
 /* Описание переменных */
 n, factorial, i: %;

 /* Ввод значения */
 input(n);
 /* Инициализация факториала */
 factorial = 1;
 i = 1;
 /* Вычисление факториала */
 do while (i <= n)
 {
   factorial = factorial * i;
   i = i + 1;
 } loop;

 /* Вывод результата */
 output(factorial);
}
```

5. Проверка числа на простоту.

```
{
 /* Описание переменной */
 num, i: %;
 is_prime: $;

 /* Ввод значения */
 input(num);

 /* Проверка на простоту */
 is_prime = true;
 i = 2;
 do while (i < num)
 {
  if num % i = 0 then
    is_prime = false;
  end_else
  i = i + 1;
 } loop;

 /* Вывод результата */
 output(is_prime);
}
```

Существуют описания синтаксиса языков программирования: формальные грамматики, и диаграммы Вирта.

Диаграммы Вирта

В метаязыке диаграмм Вирта используются графические примитивы. При построении диаграмм учитывают следующие правила:

каждый графический элемент, соответствующий терминалу или нетерминалу, имеет по одному входу и выходу, которые обычно изображаются на противоположных сторонах;

каждому правилу соответствует своя графическая диаграмма, на которой терминалы и нетерминалы соединяются посредством дуг;

альтернативы в правилах задаются ветвлением дуг, а итерации - их слиянием;

должна быть одна входная дуга (располагается обычно слева или сверху), задающая начало правила и помеченная именем определяемого нетерминала, и одна выходная, задающая его конец (обычно располагается справа и снизу);

стрелки на дугах диаграмм обычно не ставятся, а направления связей отслеживаются движением от начальной дуги в соответствии с плавными изгибами промежуточных дуг и ветвлений.

Формальные грамматики

Определение 1. Формальной грамматикой называется четверка вида:

$$G = (V_T, V_N, P, S), \quad (1)$$

где V_N - конечное множество нетерминальных символов грамматики (обычно прописные латинские буквы);

V_T - множество терминальных символов грамматики (обычно строчные латинские буквы, цифры, и т.п.), $V_T \cap V_N = \emptyset$;

P – множество правил вывода грамматики, являющееся конечным подмножеством множества $(V_T \cup V_N)^{+¿¿} \times (V_T \cup V_N)$ * ; элемент (α, β) множества P называется правилом вывода и записывается в виде α→β (читается: «из цепочки α выводится цепочка β»);

S – начальный символ грамматики, S ∈VN.

Для записи правил вывода с одинаковыми левыми частями вида $\alpha \rightarrow \beta_1$, $\alpha \rightarrow \beta_2$, …, $\alpha \rightarrow \beta_n$ используется сокращенная форма записи $\alpha \rightarrow \beta_1 | \beta_2 \vee \ldots | \beta_n$.

Описание с помощью формальных грамматик синтаксиса модельного языка приведено ниже. Грамматика имеет правила вывода вида:

P → {D1 B}
D1 → D|D1;D
D → I1: % | I1: ! | I1: $
I1 → I | I1,I
B → S1
S1 → S | S1;S|S2
S2 → I = N | I = L
S → if E then S else S end_else |do while E S loop| input(I1) | output(E) | for (S2; E1; S2) S
E → E1 | E1==E1 | E1>E1 | E1<E1 | E1<>E1 | E1>=E1 | E1<=E1
El → T | T+E1 | T-E1 | T or El
T → F | F*T | F/T | F and T
F → I | N | L | not F | (E)
L → true | false
I → C | IC | IR
N → R | NR
C → a | b | c | d | e | f | g | h | i | j | k | l | m | n | o | p | q | r | s | t | u | v | w | x | y | z
R → 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

Глава 3. Лексический анализатор

Лексический анализатор (ЛА) – это первый этап процесса компиляции, на котором символы, составляющие исходную программу, группируются в отдельные минимальные единицы текста, несущие смысловую нагрузку – лексемы.[3]

Задача лексического анализа - выделить лексемы и преобразовать их к виду, удобному для последующей обработки. ЛА использует регулярные грамматики. ЛА необязательный этап компиляции, но желательный по следующим причинам:

1) замена идентификаторов, констант, ограничителей и служебных слов лексемами делает программу более удобной для дальнейшей обработки;

2) ЛА уменьшает длину программы, устраняя из ее исходного представления несущественные пробелы и комментарии;

3) если будет изменена кодировка в исходном представлении программы, то это отразится только на ЛА. В процедурных языках лексемы обычно делятся на классы:

1) служебные слова;

2) ограничители;

3) числа;

4) идентификаторы.

Каждая лексема представляет собой пару чисел вида (n, k), где n – номер таблицы лексем, k - номер лексемы в таблице. Входные данные ЛА - текст транслируемой программы на входном языке. Выходные данные ЛА - файл лексем в числовом представлении. [3]

Таблица 1 – Входная таблица ключевых слов для ЛА

Ключевые слова	
1	%
2	!
3	$
4	for
5	do while
6	loop
7	if
8	then
9	else
10	end_else
11	input
12	output

Таблица 2 – Входная таблица ключевых слов для ЛА

Разделители	
1	:
2	+
3	(
4	)
5	=
6	>
7	<
8	<>
9	<=
10	>=
11	-
12	or
13	*
14	/
15	and
16	not
17	,
18	;
19	{
20	}

Пример для программы 1:

```
{
/*Описание переменной*/
num, i: %;
is_prime: $;

/* Ввод значения */
input(num);

/*Проверка на простоту*/
is_prime = true;
i = 2;
do while (i < num)
{
  if num % i = 0 then
    is_prime = false;
  end_else
  i = i + 1;
} loop;

/*Вывод результата*/
output(is_prime);
}
```

Ключевые слова и типы	
0	$
1	%
2	do while
3	end_else
4	if
5	input
6	loop
7	output
8	then

Разделители	
0	(
1	)
2	+
3	,
4	:
5	;
6	<
7	=
8	{
9	}

Константы	
0	0
1	1
2	2
3	false
4	true

Индикаторы	
0	i
1	is_prime
2	num

Вывод:

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

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

(3, 0), (3, 1), (3, 2),

(4, 0), (4, 1), (4, 2), (4, 3), (4, 4)

Глава 4. Синтаксический анализатор

Задача синтаксического анализатора (СиА) - провести разбор текста программы, сопоставив его с эталоном, данным в описании языка. Для синтаксического разбора используются контекстно-свободные грамматики (Ксграмматики).

Один из эффективных методов синтаксического анализа – метод рекурсивного спуска. В основе метода рекурсивного спуска лежит левосторонний разбор строки языка. Исходной сентенциальной формой является начальный 20 символ грамматики, а целевой – заданная строка языка. На каждом шаге разбора правило грамматики применяется к самому левому нетерминалу сентенции. Данный процесс соответствует построению дерева разбора цепочки сверху вниз (от корня к листьям).[4]

Пример Дана грамматика G({a, b, c, ⊥}, {S, A, B}, P, S) с правилами P :

1) S → AB ⊥; 2) A → a; 3) A → cA; 4) B → bA.

Требуется выполнить анализ строки cabca⊥.

Левосторонний вывод цепочки имеет вид: S ⇒ AB ⊥⇒ cAB ⊥⇒ caB ⊥⇒ cabA ⊥⇒ cabcA ⊥⇒ cabca ⊥ .

Нисходящее дерево разбора цепочки представлено на рисунке 1

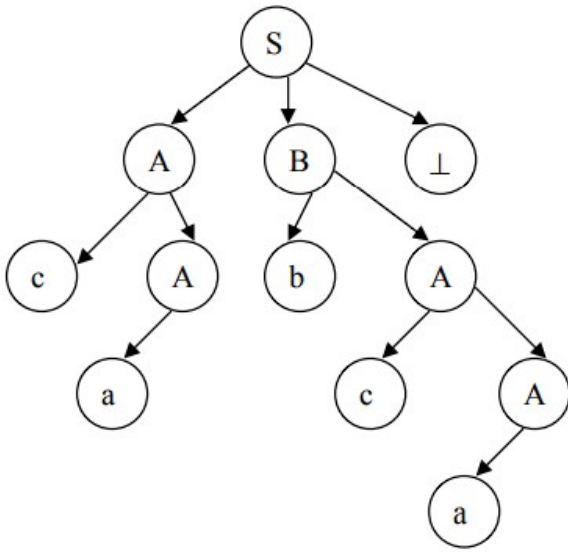


Рисунок 1 – Дерево нисходящего разбора цепочки cabca⊥ Метод рекурсивного спуска реализует разбор цепочки сверху вниз следующим образом. Для каждого нетерминального символа грамматики создается своя процедура, носящая его имя. Задача этой процедуры – начиная с указанного места исходной цепочки, найти подцепочку, которая выводится из этого нетерминала. Если такую подцепочку считать не удается, то процедура завершает свою работу вызовом процедуры обработки ошибок, которая выдает сообщение о том, что цепочка не принадлежит языку грамматики и останавливает разбор. Если подцепочку удалось найти, то работа процедуры считается нормально завершенной и осуществляется возврат в точку вызова. Тело каждой такой процедуры составляется непосредственно по правилам вывода соответствующего нетерминала, при этом терминалы распознаются самой процедурой, а нетерминалам соответствуют вызовы процедур, носящих их имена.[4]

Теории множеств FIRST и FOLLOW играют важную роль в синтаксическом анализе контекстно-свободных грамматик.

1) **FIRST(α)**: Множество терминалов, которые могут начинать строки, выводимые из α (включая пустую строку, если α может порождать её). Формально, для произвольной строки α, FIRST(α) определяется следующим образом:

- Если α является терминалом, то FIRST(α) содержит только этот терминал.
- Если α является нетерминалом и имеет вид $X_1X_2...X_n$, где X_1, X_2, ..., X_n - символы грамматики (терминальные или нетерминальные), то FIRST(α) содержит FIRST(X_1), за исключением ε (если X_1 может порождать ε), затем FIRST(X_2), за исключением ε (если X_2 может порождать ε), и так далее, пока ε не перестанет быть возможной входной цепочкой.

2) **FOLLOW(A)**: Множество терминалов, которые могут непосредственно следовать за A в выводе цепочки. Формально, для нетерминала A, FOLLOW(A) определяется следующим образом:

- Символ $ (доллар) обычно добавляется в FOLLOW(S), где S - начальный символ грамматики, и он представляет конец ввода.
- Для каждого правила вида $X \rightarrow \alpha A \beta$, где α и β - строки символов (терминальных или нетерминальных), все символы из FIRST(β), за исключением ε, добавляются в FOLLOW(A).
- Если есть правило вида $X \rightarrow \alpha A$, где A - последний символ в правой части, то все символы из FOLLOW(X) добавляются в FOLLOW(A).

Эти множества используются в алгоритмах синтаксического анализа, таких как метод рекурсивного спуска, для принятия решений о том, какое правило грамматики применить в данном контексте.

Множества FIRST и FOLLOW для моего варианта:

FIRST	
P	{
S2	a | b | c...
P1	a | b | c...
D	a | b | c...
I1	a | b | c...
I	a | b | c...
C	a | b | c...
B	}
S1	if | do | input | output | for
S	if | do | input | output | for
E	a | b | c… | 0 | 1… | not | (| true | false
E1	a | b | c… | 0 | 1… | not | (| true | false
T	a | b | c… | 0 | 1… | not | (| true | false
F	a | b | c… | 0 | 1… | not | (| true | false
L	true | false
N	0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

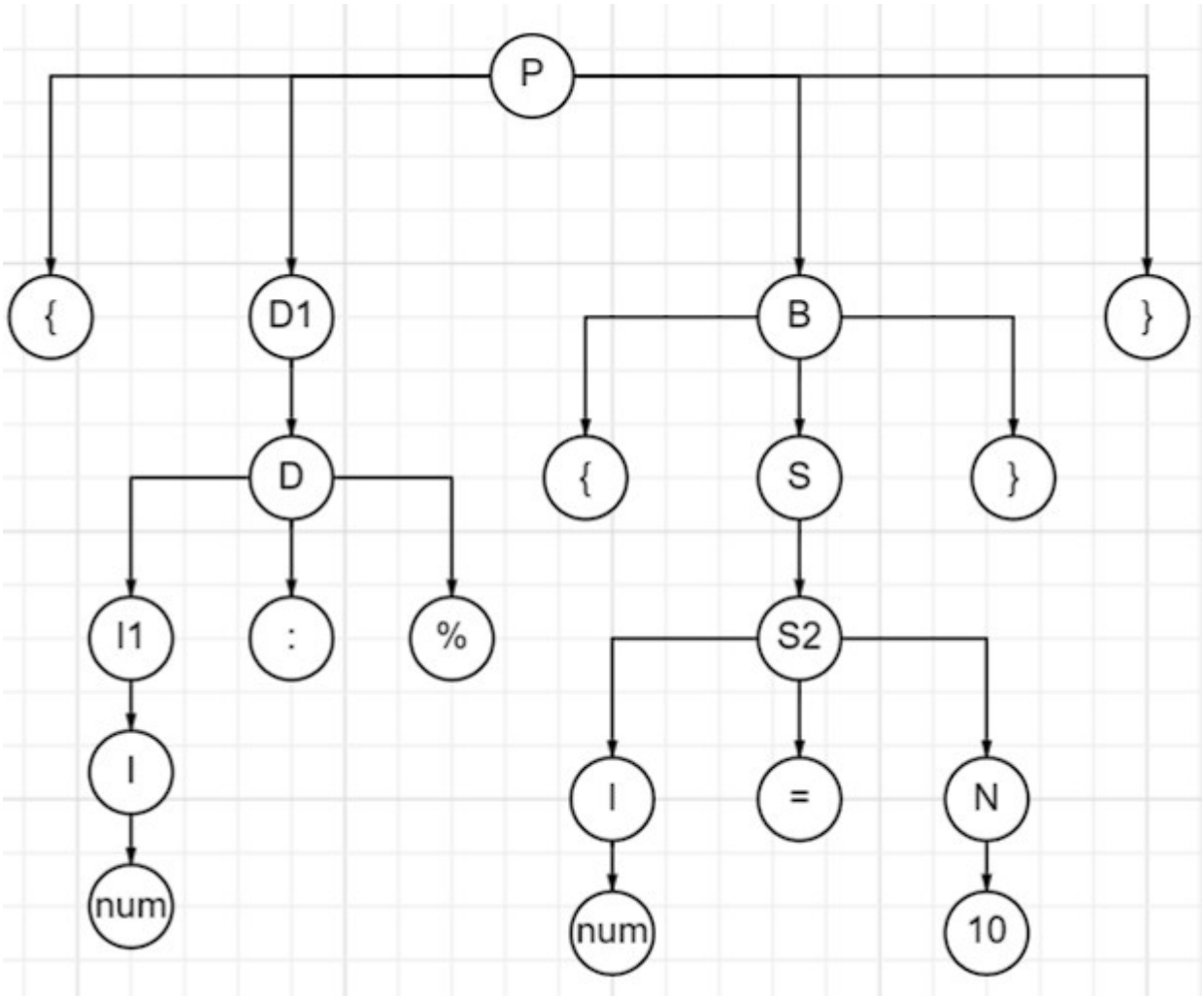
FOLLOW	
P	e
S2	} | ;
D1	} | ;
D	{
I1	: % | : ! | : $ |) | ,
I	a | b | c… | 0 | 1 | 2… | * | / | and | + | - | or | = | > | < | <> | <= | >=
C	{
B	}
S1	} | ;
S	loop | else | end_else | }
R	{
E	) | then | if | do | input | output | for
E1	; | = | > | < | <> | <= | >= |) | then | if | do | input | == | output | for
T	+ | - | or
F	* | / | and | + | - | or
L	* | / | and | + | - | or | = | > | < | <> | <= | >= | ==
N	0 | 1 | 2… | * | / | and | + | - | or | = | > | < | <> | <= | >= | ==

Разбор куска программы:

{ num : % { num = 10 } }

Стек	Входной буфер	Действие
P	{ num : % { num = 10 } }	Свертка P → {D1 B}
{D1 B}	{ num : % num = 10 }}	Выброс
D1 B}	num : % { num = 10 }}	Свертка D1 → D
D B}	num : % { num = 10 }}	Свертка D → I1 : %
I1 : % B}	num : % { num = 10 }}	Свертка I1 → I
I1 : % B}	num : % { num = 10 }}	Свертка I → num
num : % B}	num : % { num = 10 }}	Выброс
B}	{ num = 10 }}	Свертка B → {S}
{S}}	num = 10 }}	Выброс
S}}	num = 10 }}	Свертка S → S2
S2}}	num = 10 }}	Свертка S2 → I = N
I = N}}	num = 10 }}	Свертка I → num
num = N}}	num = 10 }}	Выброс
N}}	10 }}	Свертка N → 10
10}}	10 }}	Выброс
e	e	

Дерево разбора:



Заключение

В ходе выполнения курсовой работы были рассмотрены методы трансляции и алгоритмы синтаксического и лексического анализа в контексте разработки компиляторов. Были изучены основные этапы разработки компилятора модельного языка программирования, включая составление формального описания языка с использованием методов РБНФ, диаграммы Вирта и формальных грамматик.

Основные задачи были выполнены успешно, что позволило закрепить теоретические знания и приобрести опыт работы с основными алгоритмами и методами трансляции.

Разработаны и реализованы методы лексического и синтаксического анализа, а также алгоритмы сканирования и разбора текста программы на данном языке. Это позволило закрепить теоретические знания и получить практические навыки в области разработки компиляторов.

Таким образом, выполнение данной курсовой работы позволило углубить понимание процесса разработки компиляторов и приобрести опыт работы с основными алгоритмами и методами трансляции.

Список литературы

1) Ахо А., Сети Р., Ульман Д. Компиляторы: принципы, технологии и инструменты.: Пер. с англ. – М.: Изд. дом «Вильямс», 2001. – 768с.

2) Волкова И.А., Руденко Т.В. Формальные языки и грамматики. Элементы теории трансляции. – М.: Диалог-МГУ, 1999. – 62с.

3) Грис Д. Конструирование компиляторов для цифровых вычислительных машин: Пер. с англ. – М.: Мир, 1975. – 544с.

4) Жаков В.И., Коровинский В.В., Фильчаков В.В. Синтаксический анализ и генерация кода. – СПб.: ГААП, 1993. – 26с.

Приложение

1. Код лексического анализатора на языке C++:

```
#include <iostream>

#include <fstream>

#include <regex>

#include <set>

#include <vector>


using namespace std;


void lex(vector<string>& V1, vector<string>& V2, vector<string>& V3, vector<string>&
V4, const string& input_string) {

    set<pair<string, string>> keywords_and_types;

    set<pair<string, string>> separators;

    set<pair<string, string>> identifiers;

    set<pair<string, string>> constants;


    vector<pair<string, string>> patterns = {

        {"[ \t\n]+", ""},

        {"/\\*.*\\*/", ""},

        {"\\{", "LBRACE"},

        {"\\}", "RBRACE"},

        {"for", "FOR"},

        {"do while", "DO WHILE"},

        {"loop", "LOOP"},

        {"if", "IF"},

        {"then", "THEN"},

        {"else", "ELSE"},

        {"end_else", "END_ELSE"},

        {"input", "INPUT"},

        {"output", "OUTPUT"},
```

```
        {":", "DUBLE_DOT"},
        {"[0-9]+|true|false", "CONSTANT"},
        {"[%!$]", "TYPE"},
        {"(<=|>=|<|>|=|<>|\\+|-|\\*|/|and|or|not)", "OPERATOR"},
        {"[a-zA-Z][a-zA-Z0-9|_]*", "IDENTIFIER"},
        {",", "COMMA"},
        {";", "SEMICOLON"},
        {"\\(", "LPAREN"},
        {"\\)", "RPAREN"}
    };

    size_t pos = 0;
    while (pos < input_string.size()) {
        smatch match;
        for (const auto& pattern : patterns) {
            string regex = pattern.first;
            string token_type = pattern.second;
            std::regex r(regex);
            if (regex_search(input_string.begin() + pos, input_string.end(), match,
r) && match.position() == 0) {
                if (!token_type.empty()) {
                    if (token_type == "TYPE" || token_type == "FOR" || token_type ==
"LOOP" || token_type == "DO WHILE" ||
                        token_type == "IF" || token_type == "THEN" || token_type ==
"ELSE" || token_type == "INPUT" ||
                        token_type == "OUTPUT" || token_type == "END_ELSE") {
                        keywords_and_types.insert({ match.str(0), token_type });
                    }
                    else if (token_type == "LBRACE" || token_type == "RBRACE" ||
token_type == "OPERATOR" ||
                        token_type == "COMMA" || token_type == "SEMICOLON" ||
token_type == "LPAREN" ||
                        token_type == "RPAREN" || token_type == "DUBLE_DOT") {
```

```
                    separators.insert({ match.str(0), token_type });
                }
                else if (token_type == "IDENTIFIER") {
                    identifiers.insert({ match.str(0), token_type });
                }
                else if (token_type == "CONSTANT") {
                    constants.insert({ match.str(0), token_type });
                }
            }
            pos += match.length();
            break;
        }
    }
    if (match.empty()) {
        cout << "Unexpected character: " << input_string[pos] << endl;
        return;
    }
}

for (const auto& token : keywords_and_types) {
    V1.push_back(token.first);
}

for (const auto& token : separators) {
    V2.push_back(token.first);
}

for (const auto& token : identifiers) {
    V3.push_back(token.first);
}
```

```
    for (const auto& token : constants) {
        V4.push_back(token.first);
    }
}

void readFile(string& input_string) {
    string nameFile;
    cout << "Enter file name: ";
    cin >> nameFile; '\n';

    ifstream file(nameFile);
    if (!file.is_open()) {
        input_string = "File not found!";
        return;
    }

    // Считываем содержимое файла в строку
    input_string.assign((istreambuf_iterator<char>(file)),
istreambuf_iterator<char>());

    file.close();
}

void printTable(vector<string> Table, int N) {
    for (int i = 0; i < Table.size(); i++) {
        cout << "( " << N << ", " << i << ", " << Table[i] << " )" << ", ";
    }
    cout << " " << endl;
}

int main() {
    setlocale(LC_ALL, "Russian_Russia.1251");
```

```
    string input_string;
    vector<string> Table1;
    vector<string> Table2;
    vector<string> Table3;
    vector<string> Table4;

    readFile(input_string);
    lex(Table1, Table2, Table3, Table4, input_string);

    printTable(Table1, 1);
    printTable(Table2, 2);
    printTable(Table3, 3);
    printTable(Table4, 4);

    return 0;
}
```